AD-A183 750   ADA (TRADE NAME) SOFTWARE ENGINEERING EDUCATION AND   1/3
                TRAINING SYMPOSIUM (2..(U) ADA JOINT PROGRAM OFFICE
   ARLINGTON VA   C MCDONALD ET AL. 11 JUN 87
UNCLASSIFIED   F/G 12/5   NL
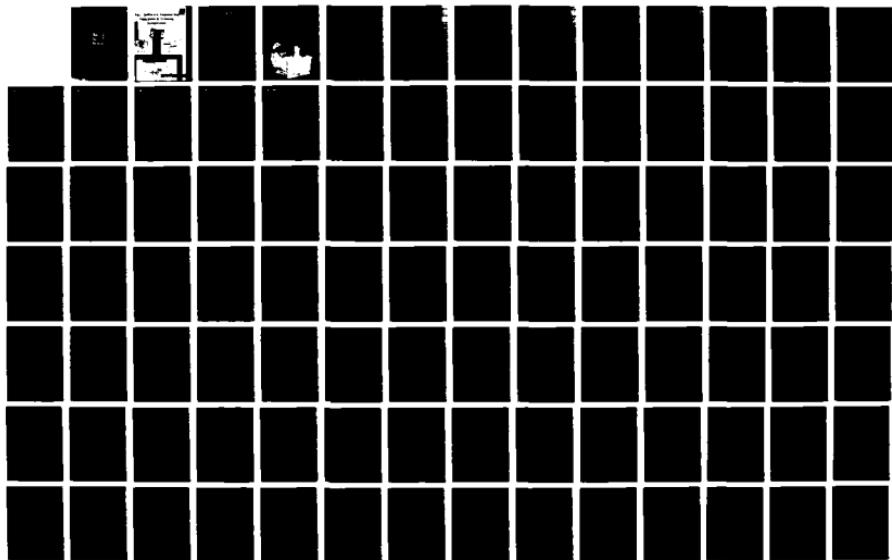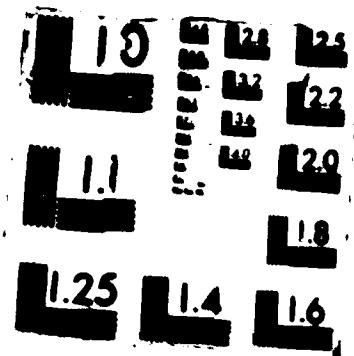
# Ada® Software Engineering Education & Training Symposium

DTIC
SELECTE
AUG 1 4 1987

INDUSTRY
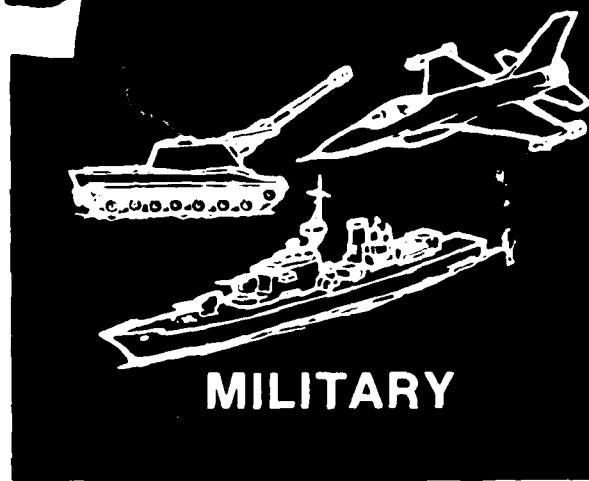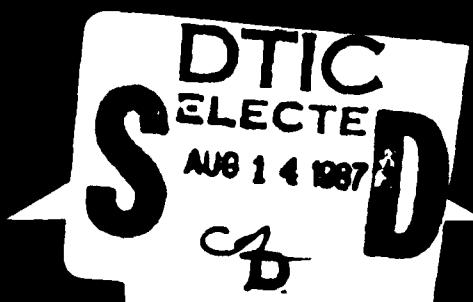
MILITARY
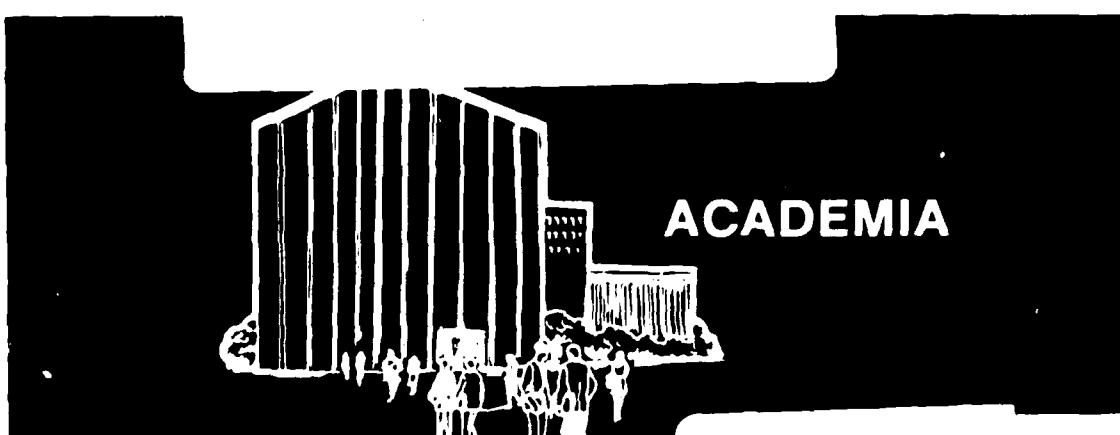
ACADEMIA

® Ada is a registered trademark of the US Government (AJPO)

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER　AD-A183750 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Software Engineering Education and Training Symposium	5. TYPE OF REPORT & PERIOD COVERED Symposium, June 9-11, 1987
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Catherine McDonald and Greg Kee, Ed.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Ada Software Education and Training Team Ada Joint Program Office, 3E114, The Pentagon, Washington, D.C. 20301-3081	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 3E 114, The Pentagon Washington, DC 20301-3081	12. REPORT DATE June 11, 1987
	13. NUMBER OF PAGES 217
14. MONITORING AGENCY NAME & ADDRESS *(If different from Controlling Office)* Ada Joint Program Office	15. SECURITY CLASS *(of this report)* UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*

Ada Programming language, Ada Training, Education, Training, Computer Programs, Ada Joint Program Office, AJPO

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This document is a collection of twenty papers presented at the Second Annual Ada Software Engineering Education and Training (ASEET) Symposium, June 9-11, 1987, in Dallas, Texas.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73　S/N 0102-LF-014-6601

Ada Software Engineering Education and Training Symposium

June 9-11, 1987
Dallas Market Center Marriott
Dallas, Texas

PROCEEDINGS OF THE SECOND ANNUAL ASEET SYMPOSIUM

Edited by :   Catherine McDonald
                  Greg Kee

Sponsored by:
Ada Software Engineering Education and Training Team
and
Ada Joint Program Office

Ada Software Engineering Education and Training Team

Mr. John Bailey

Lt. David Cook

Lt. Tony Dominice

LCDR Dave Endicott

Major Charles Engle

LtCol Rick Gross

Mr. Paul Howe

Mr. Greg Kee

Major Allan Kopp

Major Patricia Lawlis

Ms. Catherine McDonald

LCDR Melinda Moran

Major Doug Samuels, Chair

Major Randall Saylor

Major Ken Schoonover

Mr. James Tucker

Captain David Umphress

This page left blank intentionally

TABLE OF CONTENTS

Page

v

Thursday, 11 June 1987

Chairman's Forward

It is with great pleasure that I welcome you on behalf of the Ada* Software Engineering Education and Training (ASEET) Team to the Second Annual ASEET Symposium. We hope the symposium provides you the opportunity to acquaint yourself with some of the education and training materials available within the Ada community. This year we have both an academic and industry track. We're certain the information you obtain here will be extremely beneficial to you and your organization. There's a lot to learn from us as well as other people already established in academia and industry. Take this opportunity to converse with each other and exchange as many ideas as you can. We hope you have a great week in Dallas.

Major Doug Samuels, USAF
Chair, ASEET Team

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

This page left blank intentionally

Managing the Implementation

of an

Ada® Training Program

June 10, 1987

Second Annual
Ada Software Engineering Education
and Training (ASEET) Team Symposium

Presented by:

Paul Barkowitz
Harris Corporation
2101 West Cypress Creek Road
Fort Lauderdale, Florida 33309
(800) 245-6453

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

88 HARRIS Education Center

ABSTRACT

This paper details the management process used to initiate development of an Ada training program. An overview of the various issues involved is followed by a description of the implementation procedure. The setting is a training department in a commercial environment. The existing staff have various technical, teaching, and course development skills but no Ada experience. One consideration is whether to hire an Ada expert from outside or develop Ada expertise from within. Another concern is how to determine goals and objectives for the Ada training program. A discussion of the different training materials to be developed is coupled with a description of the methods used for obtaining necessary resources. Finally, means of evaluating the quality of training are discussed.

HARRIS Education Center

I would like to thank the ASEET Symposium Committee for giving me the opportunity to present to you, the Ada Programming Language training community, a description of the process I used to implement an Ada training program.

Background

Let me first give you a feel for my background and the environment in which I work. I am the Manager of Software Training for Harris Computer Systems, a 2 billion dollar a year information processing, communications, and electronics corporation. The Computer Division of Harris specializes in the manufacture and assembly of supermini computer systems for a broad customer base in the engineering, scientific, educational, and aerospace industries.

The Education Center functions as an autonomous unit within Harris and operates as a separate profit center. We are staffed with both software and hardware instructors whose backgrounds include education, the military, academia, engineering, and application programming environments. As the Software Manager of our Education Center, I oversee the development and delivery of training courses designed to handle the educational needs of our customers and internal Harris employees. My background includes a Master of Science degree and published research papers in the Social Sciences combined with 10 years of working experience in the computer industry.

Harris, in 1984, began developing Ada on its computer systems as a result of the perceived needs of our aerospace and defense customers. I found myself tasked with preparing an Ada training curriculum to support our product. As we went through the development of our Ada training program, a number of issues had to be addressed and key management decisions had to be made. These included:

- Finding Ada Expertise

- Setting Goals and Objectives for the Ada Training Program

- Determining the Number and Length of the Courses

- Producing the Necessary Training Materials

- Judging the Quality of our Ada Training

Finding Ada Expertise

Our first problem was locating Ada expertise. The Harris Ada Product Management team consisted of Ada marketing experts, and our Software Development staff had a number of Ada syntax experts, but nowhere did we have a true Ada practical coding expert - someone who had spent several years generating extensive Ada code in an applied environment.

The issue became, do we hire an Ada technical expert from outside of our company or

develop the expertise from within our training organization? Hiring an Ada expert would allow us to begin course development almost immediately, while developing our own Ada expert would be a lengthy process. Back in 1984, we quickly realized that true Ada experts with years of industry experience were few and far between. Most Ada "experts" of the time were really Ada syntax experts, and they would have been expensive to hire. Further, just because they were Ada experts was no guarantee that they would be Ada *educational* experts who could successfully develop and deliver Ada training courses.

We reached the decision to develop our own Ada expert. As much as we wanted to deliver Ada training as soon as possible, we realized that the best approach was a more cautious path of development. As a training organization for a manufacturer of computer hardware and Ada software, more than just sales of future training hinge on our success. The quality of our training directly reflects upon the quality of our hardware and software products. We had to develop our Ada training program as part of an overall responsibility of service to an existing customer base.

Within our Education Center we had a complete staff of instructors with proven programming and educational knowledge. We had an Ada Product Management team who could steer our course development and a Software Development team to provide us with technical guidance. As management, we were willing to take one of our instructors and have him devote months of background research and programming in Ada prior to beginning the development of our first Ada course.

Setting Goals and Objectives

Our next challenge was to determine the direction our Ada training should take. Fortunately, Joe Dangerfield, Vice Chairperson of AdaJUG, serves as Harris' Product Manager for Ada. We turned to him for guidance on setting the goals and objectives for our Ada training program. Joe reviewed our course outlines and materials and was the initial steering force behind our Ada course development.

Once we started delivering Ada training, we became directly involved with the users of Ada, and we started relying on them to guide us on what they needed to know. Recently, we have begun development of a peer review relationship with an academic institution heavily involved with Ada implementation at the Federal level. We are providing them with copies of our training materials, and they are giving us critical feedback allowing us to insure that our training program meets the needs of the Ada community.

Determining the Number and Length of the Courses

Our next decision concerned how many courses would comprise our Ada training curriculum and how many days these courses would last. We were cognizant that as we made our training program more in-depth and

cost (both in time and dollars) to the student. Given that we operate in a vocational setting, we realize that our customers are limited in the amount of time they can dedicate for their employees to attend training classes. To minimize our customers' financial expenditures, we decided to cover as much information as possible in the least amount of time. To do this we had to make an important assumption.

We assumed that our students would be programmers experienced in a high level language and thus should be familiar with software engineering principles. With that as a prerequisite to our first Ada course, instead of preparing a separate introduction to software engineering principles, we integrated this discussion into our presentation of programming within Ada.

Rather than cover all the features of Ada in one course, we decided to create separate courses for the introductory and advanced topics. Further, we wanted to separate the Harris-specific implementation of Ada from the more general programming features. Finally, we saw the need for an advanced workshop to provide students with an environment for extensive exploration of specialized programming applications, the large-scale production of maintainable code, and the life cycle support of Ada software.

The result of these decisions is a 4-course curriculum. Our first course is a 1-week Introduction to the Ada Programming Language. This is followed by a 1-week Advanced Ada course. A 3-day course is provided for those interested in the Harris Ada Programming Support Environment (HAPSE®), and our most advanced course is a 1-week Ada Programming Workshop.

Producing the Necessary Training Materials

The key educational tool to support our live lecture delivery of information is the Student Guide. Produced as a learning tool, as opposed to a reference document, the design of this guide follows adult learning concepts. Information is presented in a manner that facilitates learning.

Lectures are followed by laboratory sessions designed to reinforce, in a practical setting, the theoretical subject matter. Students are given exercises of increasing difficulty, and those who finish a lab ahead of schedule are given optional exercises to complete. Since approximately 50% of each course is spent in the lab, equal emphasis was placed on student exercise development and student guide creation.

Technical documentation is provided through use of the *Ada Programming Language*

---

* HAPSE is a trademark of Harris Corporation.

Reference Manual (ANSI/MIL-STD-1815A). Additionally, each student is given a copy of Grady Booch's, *Software Engineering With Ada.* From this text, the instructor suggests supplemental readings to complement in-class presentations.

Every course contains a companion guide for the instructor to accompany the student guide. This Instructor Guide promotes consistency across course presentations. It is also used by instructors preparing to teach a course for the first time. Besides pointing out important topics for discussion, the Instructor Guide contains preparation notes, optional activities, cross-references between the Student Guide and technical documentation, and solutions to student exercises.

Judging the Quality of our Ada Training

One means of determining effectiveness of training is through the measurement of student progress. Instructors evaluate practical knowledge through the student's performance on laboratory exercises, while gains in conceptual knowledge are measured through in-class testing.

More importantly, we rely on student feedback as an important method of judging the quality of our training. At the conclusion of each course, students are asked to evaluate the training on a number of dimensions, including quality of the instructor, the course materials, and the learning environment. Our course critique form combines numerical ratings with open-ended items. Students are encouraged to suggest additional topics for discussion and changes in course content or emphasis.

We have been delivering Ada training both domestically and internationally for more than two years. Our students have included both users of Harris computers as well as those needing Ada training on other, non-Harris, equipment. With this diversity of experience, we have received feedback from customers in a wide variety of applications.

Members of our Ada Software Development team have taken our courses and have given us technical guidance. Additionally, many of our field analyst force and our Ada Product Marketing team have been through our Ada curriculum and have provided us with suggested changes to meet customer needs.

Our relationships with those in academia working at the forefront of Ada continue to provide us with a critical external peer review of our Ada curriculum. Although our instruction is based in a vocational setting, it is important to us to maintain an academic standard of excellence.

At Harris, we have found that development of an Ada training program is an ongoing process. As educators of Ada, we must keep current with the latest technological advances and be responsive to the changing needs of our customers.

BIBLIOGRAPHY

Booch, G. *Software Engineering With Ada, Second Edition.* Menlo Park: Benjamin/Cummings Publishing Company, Inc., 1987.

Reference Manual for the Ada Programming Language, Ada Joint Program Office, Department of Defense, Washington, D.C., February 1983, ANSI/MIL-STD-1815A.

This page left blank intentionally

The SEI Ada Training Guide: Status and Motivation

Priscilla J. Fowler
John H. Maher, Jr.
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania
April 30, 1987

Abstract: The Ada[1] Training Guide is a project of the Software Engineering Institute's Training and Transition Methods Program. The Ada Training Guide will facilitate the transitioning of Ada by giving direction for the selection and evaluation of training in Ada and related topics. The first version of the guide is targeted for System Program Offices (SPOs), where management as well as technical staff must have training in Ada and its implications. Later versions will add material for software project managers, training managers, and software engineering practitioners. The guide will serve as a prototype for future guides addressing training in other software engineering topics, and is designed to complement related activities of the Ada Joint Program Office.

This paper describes the proposed guide and the impetus and process for its creation.

The Need for the Ada Training Guide

A Look At Ada Transition Work To Date

Getting Ada into routine use requires major training and education efforts. It also requires a massive technology transition effort to address both the technological and the human changes which are required. Considerable effort has been expended over the last several years to address the technological aspects of Ada, including compilers, environments, and demonstration projects. Significant effort has also been spent in dealing with the need for Ada training and education, and the need for information about Ada resources. The Ada Joint Program Office in particular has sponsored many substantial efforts, including the Ada Information Clearinghouse, the the Catalog of Resources for Education in Ada and Software Engineering (CREASE) [5], and the Ada Software Engineering Education and Training (ASEET) Team [3]. The broad training needs assessment by the Commission of the European Communities (CEC) [7], completed in 1984, is also worthy of note.

These efforts are to be commended as a major step towards effecting Ada transition. There is now broad awareness of Ada at all levels, as well as an understanding of curriculum needs for software practitioners who will use Ada.

What is needed next is work to assist specific organizations in using Ada in specific projects [11, 12]. That is, what is needed is technology transition skills *within each organization* [4, 9, 10, 15, 16] which will adopt Ada, whether by choice or by mandate. Each organization which will use Ada must be able to select appropriate training, and plan effective Ada insertion strategy without the help of experts.

Training as an Ada Insertion Mechanism

In order for Ada to become broadly integrated into the MCCR software community, software practitioners skilled in the use of Ada must be readily available. Training courses must therefore also be available. In addition, training which is available must be carefully selected if it is to serve to facilitate adoption of Ada by an organization, and not just enhance individual skill sets.

To date, Ada curricula [5] have offered generic solutions to the question of who needs what skills. Usually they have targeted typical job categories within software development projects. And while the study by the CEC has attended to the training needs of managers, commercial offerings of Ada courses often omit managers or provide them with inappropriate content or educational approach.

Useful Ada curriculum design must address all affected populations. Practitioners and their project managers make up the first of these. Officers and executives who manage groups of projects, or who make major policy or resource decisions are a second important group. And finally, program office personnel, who more and more frequently are working on contracts involving Ada, make up a third, and very key, group. Without support from management [14, 16] and technology transition partners such as program office personnel, training of practitioners will be of limited value, since they will not be able to use what they have learned.

The Ada Research Centers proposed by ASEET may be able to address curriculum design and development needs for all these groups in the future. In the meantime, the best strategy is to select judiciously from courses already available. It is rare that the person most familiar with technology such as Ada also has education and training skills. Thus the rationale behind the Ada Training Guide is to provide useful procedures, based on sound educational principles [2, 13], for the selection and evaluation of Ada training, where expertise in training and education is not readily available.

Content of the Guide

The Ada Training Guide will contain four sections. The first section will provide procedures for Ada training needs assessments. The second will suggest approaches to selecting training.

The third will review useful evaluation procedures. And the fourth will discuss strategies for Ada technology insertion. Each of the sections is described briefly below.

Training Needs Assessments

Instructional designers know that good training is wasted on students whose training needs don't match the ones for which the course was designed. For instance, an Ada course for programmers who are already familiar with Pascal will be quite different from one designed for Fortran programmers. A course for project managers will not provide enough detail on Ada packages and generics for a designer, and will discuss topics, such as Ada's impact on cost estimation methods, which do not interest a designer at all. It is therefore extremely important to determine, before a course is acquired or developed, what potential students' needs are for training.

Questions such as these can help define student types:

1. What groups (categories) of people in your organization need to be trained?
2. At what organization level?
3. What educational background does each group have?
4. What experience does each group have?
5. What are the major job functions in each group?
6. What are the supporting/interfacing job functions in each group?
7. *How critical are the decisions made by people in this job function?*
8. How critical is the work performed by people in this job function?
9. In each case, how will Ada impact the job function?
10. In each case, what categories of information must the job holder know about Ada?

The above questions can help determine appropriate training content. Other factors, such as numbers of students needing the training, and frequency of that need, must also be considered. Training presented too early or too late may largely be wasted effort. Some examples of non-content factors are:

1. An immediate need for training, versus ongoing training requirements
2. Organizational issues, such as quality of management support
3. Budget
4. Policy, such as days per person allotted each year for training, and student selection criteria
5. Location, including facilities needed such as classroom space, audio-visual equipment.

Training Selection

Once needs for each student group have been collected and analyzed, these can be used as selection criteria. Sources such as CREASE [5], vendor brochures, and vendor demonstrations

at conferences can provide course descriptions. Selection criteria can then be mapped onto course descriptions, and more information gathered as necessary. When there are no matching courses, the criteria can be used to prepare a draft specification for developing an original course.

The Ada Training Guide will provide structured selection procedures and example sets of criteria for some typical student groups. The CEC study [7] has an excellent set of descriptions of student groups typical of software development organizations; the Ada Training Guide will reuse as much of that as possible, and will add descriptions of other typical DoD MCCR student groups, such as SPO personnel.

Training design and development is generally labor intensive as well as expensive. One hour of class time -- lab session, lecture or exercise -- can take as much as 100 hours of course development time [1]. Since this is the case, the guide will provide references to materials useful in specifying or planning for course development efforts.

Training Evaluation

Once a course has been selected, it must be evaluated. A good way to evaluate a course before sending students to it is to check the experience of organizations who have already sent students. Reputable training vendors will supply references. Even a brief telephone interview with such a reference can elicit helpful information. The questions which follow can help structure the interview:

1. How many students has the reference sent to the course?
2. Was the background of those students comparable to that of your potential students?
3. How did the job performance of students change as a result of the training?
4. Was the instructor knowledgeable? What was his or her background?
5. Was the course lively, involving, interesting?
6. Were the materials provided easy to use, helpful?
7. Were the visuals easy to read? Were copies provided?
8. Was reference material such as an annotated bibliography provided?
9. Have students used the reference material on the job? How?
10. Were there lab sessions? Describe these.
11. If this was a course for software engineers, were software tools used for "hands-on" exercises? Will those tools be the same as or similar to tools you are using or plan to use?

It is also useful to ask questions of the training vendor. Some suggested topic areas are listed here:

1. How many offerings of this course have you given?
2. To whom? Types of corporations or agencies sending students, and typical student education and experience background should be described.

3. How are instructors trained, certified?

4. Is the source of course content expertise the instructor or course developer? If not, what source was used?

5. What text is used?

6. What data can you provide to show how effective the course is?

The evaluation procedures described above are adequate for course evaluation if only a small number of students will attend a course selected. If a large number will attend, or smaller groups will attend periodically, it is worth investing in actual course attendance as a final step in evaluation. The student attending should be typical of the group that will eventually attend the course, and should be willing and able to comment on both content and, with guidance, on pedagogy. Making this initial effort is well worthwhile: spending tuition money and time away from the job for one person is much cheaper than for ten people.

Strategies for Technology Insertion

The best training will do little good if not placed in the context of an effective technology insertion strategy [14, 16]. In the case of Ada, the DoD's many dissemination mechanisms have created awareness of the technology, and its mandates have created impetus to consider Ada adoption. However, it is helpful to understand how an organization should make plans to adopt Ada. If, prior to training, an Ada compiler and a set of support tools have been identified, and local design and coding standards defined, the training can be selected to match. If such is not the case, it is necessary to time the training for most students *after* these materials have been prepared [8, 15].

Thinking in terms of transition, with training as part of the effort, can lead to other helpful strategies. For example, which student groups should be trained first? How soon can those groups use their training? It may make sense to train managers and planners early in a project, and train programmers only after software architecture is complete.

The guide will suggest transition strategies, and provide some typical scenarios as examples.

Design Philosophy

The design of the Ada Training Guide will be guided by these thoughts:

1. The guide should be as succinct as possible, and organized to provide ready access to information. The initial version will be paper-based. An alternate form of delivery, perhaps PC-based, is possible in the later versions.

2. The guide should be written by a team, representing:

    a. Ada expertise, as required by each typical student group

    b. instructional system design and evaluation expertise

    c. members of the targeted student groups

    d. software engineering expertise.

3. The guide should be tested by collaborative users, and revised accordingly.

4. The guide should be prototyped, that is, the initial version of the guide should be directed to one student group.

Status and Initial Plans

The SEI will provide both Ada and instructional design expertise. A System Program Office will be sought as a collaborator to provide additional team members with both acquisition and Ada development expertise. This team will work to produce a prototype version of the guide, with the initial users being program office personnel.

These initial users have been selected because their training needs have not been addressed by typical Ada curricula. In addition, this group is a particularly influential group within the DoD, able to influence Ada adoption in general, and Ada training in particular, especially among contractors [6].

The prototype will tell us how well the guide works. As we test the prototype, we can also determine the answers to these important questions:

- Is our test group typical of all groups of that type?
- Must the guide be redone for each student group, or can we include all typical student groups in one guide?
- Must the guide be domain-specific? That is, must it be provided in different versions for Ada for flight simulators versus Ada for missiles?
- How much effort is required to use the guide? What errors are avoided by the use of it?
- Most importantly, are people's training needs more precisely targeted and are courses selected more appropriately as a result of using the guide?

References

1. Advanced Technology. Nonpersonal Studies and Analysis Services for Assessment of New Training Technologies. F41689-84-C-0012, Air Force Air Training Command, Washington, D.C., June, 1985.

30 April 1987, 13:41

2. HQ ATC/DAPE. Instructional System Development. AF Manual 50-2, Department of the Air Force, Washington, D.C., May, 1979.

3. Institute of Defense Analysis. DoD Ada Software Engineering Education and Training Plan. Final Report P-1919, Ada Joint Program Office, Arlington, VA, January, 1986.

4. Bridges, William. "How to Manage Organizational Transition". *Training* (September 1985), 28-32.

5. IIT Research Institute. Catalog of Resources for Education in Ada and Software Engineering (CREASE), Version 3.0. MDA903-83-C-0306, Ada Joint Program Office, Arlington, VA, May, 1985.

6. Foreman, John, and Goodenough, John. Ada Adoption Handbook: A Program Manager's Guide. Software Engineering Institute, Pittsburgh, PA, May, 1987.

7. Hummel, H., Nast, M., Uthke, E., Dowling, E.J., Glynn, J.G., Thomas, R.J., Goldsack, S. J. Training Concept for the Cost-Effective Development of Reliable Software Using the Programming Language Ada: Final Report, Phase 2. Commission of the European Communities, Brussels, Belgium, September, 1984.

8. Jackson, Conrad N. "Training's Role in the Process of Planned Change". *Training and Development Journal* (February 1985), 70-74.

9. Kanter, Rosabeth Moss. "Change Masters and the Intricate Architecture of Corporate Culture Change". *Management Review 63*, 6 (October 1983), 18-28.

10. Leonard-Barton, Dorothy, and Kraus, William A. "Implementing New Technology". *Harvard Business Review* (November-December 1985).

11. Myers, Ware. "Ada: First Users--Pleased: Prospective Users--Still Hesitant". *Computer 20*, 3 (March 1987), 68-73.

12. Rogers, Everett. *Diffusion of Innovation.* Free Press, New York, 1983.

13. Romizowski, A.J.. *Designing Instructional Systems: Decision Making in Course Planning and Curriculum Design.* Nichols Publishing, New York, 1981.

14. Svoboda, Cyril P., and Sayani, Hasan H. Management and Education: Critical Factors in Technology Utilization. Proc. National Conference on Methodologies and Tools for Real-Time Systems, 1986.

15. Ebenau, R.G. Report of the Group on Training as a Technology Transfer Vehicle. Proc. IEEE 1983 Workshop on Software Engineering Technology Transfer, 1983, pp. 6-8.

16. Peters, L. Report of the Group on Behavioral Aspects of Software Technology Transfer. Proc. IEEE Report of the Group on Behavioral Aspects of Software Technology Transfer, 1983, pp. 11-14.

30 April 1987, 13:41

This page left blank intentionally

13

Ada* from a Management Perspective
by
Major Charles B. Engle, Jr.
&
Lieutenant Anthony R. Dominice

There has been a great deal of debate as to whether or not Ada is the solution to the software crisis identified by the DoD in the middle 1970's. Ada's various features and its integrated facilities make it tempting to argue that it is the panacea that many people would have you believe that it is. However, Ada is NOT the solution to the software crisis; Software Engineering can make that claim.

Software Engineering is an engineering discipline that is still in its infancy. The very principles upon which it is based are still subject to debate. In fact, a concise definition of the term "Software Engineering" has not been generally agreed upon. The Software Engineering Institute initially decided not to even try to define the term; instead they accepted the term as "axiomatic." But while not everyone can agree on what software engineering really is, there are few that dispute its value or its potential. By applying time-tested engineering principles to the management and development of software, we gain the ability to make time-critical decisions, to develop monetary and time metrics, and to manage a software project from its inception to its completion.

These ideas are not new; they have been around for at least 15 years. But our ability to use these ideas, and the methods and principles which support them, has been restricted by our lack of adequate tools. In particular, we lacked a tool with which we could realize our designs. Traditional programming languages were just not well suited to the software engineering revolution because they forced us to translate our "clean" design concepts into a few rigid language structures. The result was an abstraction of our design which was itself an abstraction of the problem. Thus, the tool which we used to solve the problem introduced added, unnecessary complexity caused by the lack of expressitivity of the implementation language.

Ada is the programming language which was itself engineered to meet this need. Starting with a set of requirements, proceeding through several design iterations until it became an ANSI standard, Ada was meant to be a TOOL with which we could directly implement the software engineering principles and methodologies that we had already developed. This is not

* Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

to say that Ada has any new or unique features; for the most part it doesn't. But Ada consists of the best features from several different languages which have been integrated to provide the user with a TOOL that is needed to implement software engineering.

In general, DoD software has certain characteristics which can be summarized as: 1) they are large, 2) they are complex, 3) they are long-lived, 4) they have a high demand for reliability, 5) they have real-time constraints , and 6) they have size constraints [1]. In the past, the DoD has suffered from certain factors which have adversely affected its software. Among these factors are : 1) ignorance of life-cycle implications, 2) lack of standards, 3) lack of methodologies, 4) inadequate support tools, 5) poor software management, and 6) lack of trained software professionals [2,3].

The traditional approach to software was that it was a necessary evil which was required to make the hardware operate. Another approach was that it was a "black art" which was done, but which decent people didn't talk about. Another traditional approach was to select some known gurus and magicians and place them in a dark room until they had developed what was needed. In all of these approaches, the DoD was forced to confront its fundamental problem: its inability to manage the complexity of its software systems [4]. This was due in large measure to its lack of a disciplined, engineering approach.

The establishment and application of sound engineering, regardless of the field, is predicated upon a hierarchy as outlined by Dr. Charles McKay [5]. First, the underlying concepts of the discipline must be identified. Principles are then developed which support the concepts. From these principles, models are created. These models give rise to various methodologies, which then cause the development of standardized tools. Collectively, these tools become an environment upon which the practitioner can rely and in which he can gain skill. When this environment is combined with standards, guidelines and practices, computational products which are correct, modifiable, reliable, efficient, and understandable throughout the life cycle of a system can be developed and supported.

A programming language is a software engineering tool. It expresses and executes design methodologies. The quality of a programming language for software engineering is determined by how well it supports a design methodology and its underlying concepts, principles, and models. Traditional programming languages were not engineered when they were designed. They have lacked the ability to express good software engineering concepts and have thus acted to constrain our use of software engineering. Ada, however, was engineered, and its rich set of constructs and features were integrated, to support sound software engineering. The Ada language embodies the same concepts, principles, and models as software engineering and thus supports several methodologies. Ada is the best tool (programming language) for software engineering currently available.

Some of the generally recognized principles of software engineering are 1) modularity, 2) abstraction, 3) localization, 4) information hiding, 5) completeness, 6) confirmability, and 7) uniformity. For a complete discussion of these principles and their definitions, the reader is referred to Grady Booch's text, Software Engineering with Ada [6].

A listing of the major features of the Ada language, in no particular order, is 1) standardization, 2) readability, 3) program units, 4) separate compilation, 5) packages, 6)

strong typing, 7) typing structures, 8) data and operation abstraction, 9) tasking, 10) exceptions, and 11) generics [7]. An exhaustive examination of how each of these features of Ada attacks a particular problem and/or supports sound software engineering would be too lengthy for this paper. Instead, we'll focus on a few of these features as examples of the strong support that Ada provides for software engineering.

Ada is an precise standard as defined in ANSI/MIL-STD-1815A [8]. It can have no subsets or supersets. Conformance to this standard is zealously guarded by the Ada Joint Program Office, whose main function is the promotion and use of Ada. Compliance is assured by the use of a trademark for the name Ada. In order to be able to use the trademark, compiler developers must pass a battery of programming tests called the Ada Compiler Validation Capability. This standardization allows for portability and it promotes reusability. But perhaps the most important result of standardization is that it shifts the focus of software from the mundane to the important. The programmer can concentrate on solving the problem, not on nuances of portability.

Ada "programs" consist of several program units. Each of these program units consist of two parts: a specification and a body. The specification defines the interface between program units; this is "what" the program unit does or can do. The body defines the implementation of the program unit; this is "how" the program unit does it. The clear separation of the "what" from the "how" allows the programmer to channel his thoughts into the problem at hand. The programming system is designed with specifications and implemented with bodies. At all times there is a clear distinction between architecture and implementation.

Separate compilation is another powerful feature of Ada. This capability is to be differentiated from independent compilation allowed in other languages [9]. Separate compilation allows the programmer to separately compile his program units, but guarantees and enforces the interfaces across compilation units. Not only are program units separately compilable, but the specification of any program unit is separately compilable from its body. In this manner, Ada realizes not only a logical distinction between architecture and implementation, but a physical distinction as well! Separate compilation allows the development and testing of independent software components, thereby encouraging us to reuse components and keep our investment. The human effort in software development need not be disposable any longer.

Packages are the fundamental feature of Ada which allow for a change in mindset. Program units which are logically related can now be "packaged up" and be placed in one physical space. This allows us to define and test reusable software components and resources. Packages are an architecture-oriented feature. They place a wall around resources which can then be selectively exported to the user. They may contain local resources which are not made available to the user. As such, packages directly support abstraction, information hiding, modularity and localization.

Data abstraction is a powerful means of combining "raw material" to form higher level structures. There are various levels of abstraction. Each level enforces an abstraction on a higher level structure. Data abstraction extracts the essence of an idea without becoming concerned about the details of the implementation. In fact, a truly abstract data type requires the ability to prohibit the use of its implementation details by higher level structures. This promotes understandability by allowing the reader to focus on the idea and not the details. It also

promotes modifiability by preventing a using program from relying upon the details of implementation, so that changing the manner in which the data is represented cannot have any effect on the user's program. In Ada, data abstraction is provided by private types which directly implement information hiding.

Tasks are Ada program units which act in parallel with other program units. This allows the programmer to directly express parallel algorithms in a natural manner. It also takes advantage of the move toward parallel hardware architectures with their gains in fault tolerance and distributed systems. The ability to express parallel actions directly and naturally in the language reduces the additional complexity which would otherwise be required to express parallel actions in a sequential manner.

It must be recognized that in today's large, complex systems, errors will occur in both the hardware and the software. Real time systems must be able to tolerate these errors and continue to operate in a degraded mode. Traditional languages lack specific features for dealing with error situations, yet reliability and safety must be engineered into a programming system. Exceptional situations such as these are handled by another of Ada's constructs, the Exception Handler. Ada provides a facility to separate the exceptional situations from the normal situations providing increased readability. When an exceptional situation, which need not always be an error, occurs, Ada can trap the exception and either correct the problem in some manner, or propagate it to the next higher level. By providing control over exceptional situations, Ada increases reliability and reduces complexity.

Ada also provides a generic facility. A generic is a tailorable template for a program unit. Generics are program units that may be "parameterized" by objects, types, or subprograms. This allows us to increase software reusability by an order of magnitude. Generics reduce the size of the program text and increase the reliability of the system by allowing reuse of known reliable components.

In the software life cycle, Ada provides large gains in productivity. We shall briefly examine some of the ways that Ada provides support for the software life cycle, but will not provide attempt to discuss each phase of the life cycle completely.

In the design phase, Ada features support most architectural design methodologies, such as top-down, bottom-up, or some combination of these approaches. A design can be expressed directly in Ada by using Ada as a Program Design Language (PDL). This allows us to enforce the design interfaces because of the use of a compilable PDL. Further, Ada features are rich enough that they reduce the need to squeeze the design into the meager features afforded by most programming 'anguages. Finally, Ada's standardization brings a much higher level of predictability because of the known semantic meaning of the Ada language itself and because of the use of existing Ada software components. Reusability of Ada components also provides excellent support for rapid prototyping.

Ada is also helpful in the coding phase because its features insure that the original design is not violated. Additionally, the use of Ada as a PDL reduces the amount of coding activity. Readability of the Ada code also promotes greater productivity.

In the component testing phase, Ada is extremely useful. The ability of Ada to support separate components allows for more effective testing. Further, Ada's exception features allow us to "build-in" testing facilities.

When we reach the system testing and integration phase, Ada has greatly reduced our workload. The use of Ada as a PDL will have already enforced our interfaces, so these are necessarily correct. Thus, more effective time can be spent in testing the system as a whole, rather than fixing small integration errors.

The biggest payoff for the use of Ada, however, comes in the maintenance phase. The use of proper software engineering techniques throughout the design and implementation phases, combined with the readabliity of Ada, should reduce the overall maintenance costs significantly. Time alone will tell the tale.

[This paper is a written form of the talk on this same
 subject given by the authors in early December at the
 Pentagon and, privately, to the Under Secretary of the
 Army.]

Bibliography

[1]   D. A. Fisher, "A Common Programming Language for the Department of Defense - Background and Technical Requirements," Institute for Defense Analyses, Report P-1191 (June 1976): 19.

[2]   M. T. Devlin, Introducing Ada: Problems and Potentials, USAF Satellite Control Facility (unpublished report), 1980, p. 2.

[3]   G. Booch, Software Engineering with Ada, Benjamin/Cummings Publishing Company, Second Edition, 1987, p. 9.

[4]   Ibid, p. 28.

[5]   C. McKay, Presentation at the First Annual ASEET Symposium, Orlando, FL, June, 1986.

[6]   Booch, op. cit., pp. 31-35.

[7]   Reference Manual for the Ada Programming Language, United States Department of Defense, Ada Joint Program Office, February 1983.

[8]   Ibid.

[9]   N. H. Cohen, Ada as a Second Language, McGraw-Hill Book Company, 1986, p. 422.

COMPARING DESIGNS:
A METHODOLOGY FOR TEACHING SOFTWARE ENGINEERING

Putnam P. Texel
TEXEL & COMPANY

Abstract: This paper describes an approach to teaching software engineering that was discovered quite accidentally and has proved to be an invaluable pedagogical tool. The paper describes the approach and its use in seminars for management and software engineers.

1. INTRODUCTION

This paper describes an approach that is used to train software engineering with Ada that has proved to be quite successful. The approach consists of a workshop centered on comparing multiple Ada designs designed to the same statement of requirements. The workshop has proved to be a valuable pedagogical tool in training software engineers. Because of that success, a decision was made to "port" part of the exercise to a technical management seminar. The results were equally successful.

Section 2 describes the background of this work. Sections 3 through 8 describe the technical details of the workshop: Section 3 contains the Statement of Requirements; Sections 4 and 6 contain two Ada designs, while 5 and 7 summarize the class evaluation of the designs with respect to the principles of software engineering; Section 8 shows how dependency diagrams are a useful tool for presenting layers of abstraction.

Finally Section 9 describes the main benefit for management and Section 10 implores those individuals currently involved in teaching software engineering with Ada to consider the inclusion of such workshops in future seminars.

2. BACKGROUND

TEXEL & COMPANY is under contract to General Dynamics to provide a software engineering curriculum based on Ada*. A portion of the resulting curriculum is shown in Figure 1. Part of TEXEL & COMPANY's responsibility is to design, implement and test teach each course in the curriculum. During the design of the Preliminary Design sequence it was apparent that simply talking about the principles of software engineering would not provide the proper level of understanding of Ada's support of these principles.

---

*Ada is a registered trademark of the US Government (AJPO)

MANAGEMENT TRACK

Ada IMPACT ISSUES → Ada : NON-TECHNICAL MANAGEMENT → Ada: BIDS/PROPOSALS → Ada : SDP

Ada IMPACT ISSUES → Ada : TECHNICAL MANAGEMENT → Ada: BIDS/PROPOSALS

SOFTWARE ENGINEERING TRACK

PRELIMINARY DESIGN with Ada - I → PRELIMINARY DESIGN with Ada - II

PRELIMINARY DESIGN with Ada - I → DETAILED DESIGN with Ada

PRELIMINARY DESIGN with Ada - II → DETAILED DESIGN with Ada

DETAILED DESIGN with Ada → Ada CODING - I → Ada CODING - II → REAL TIME ISSUES
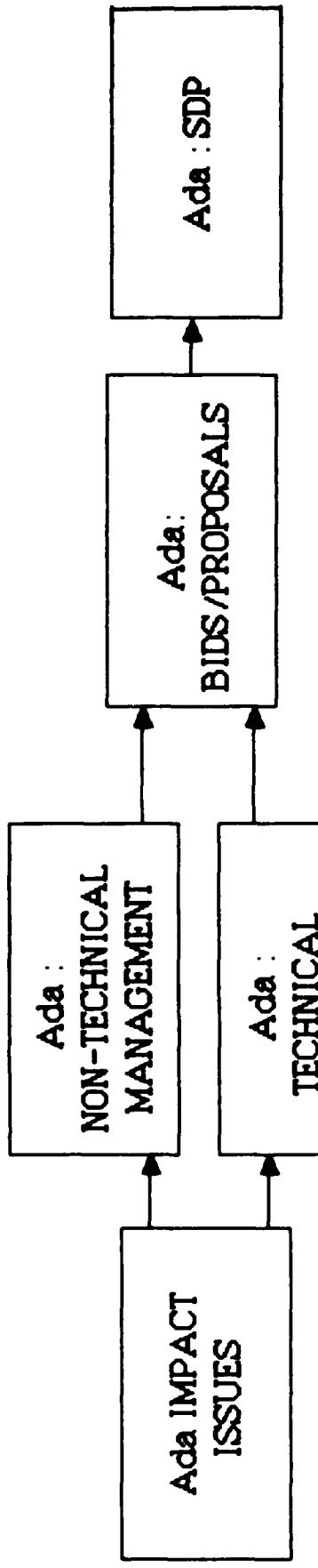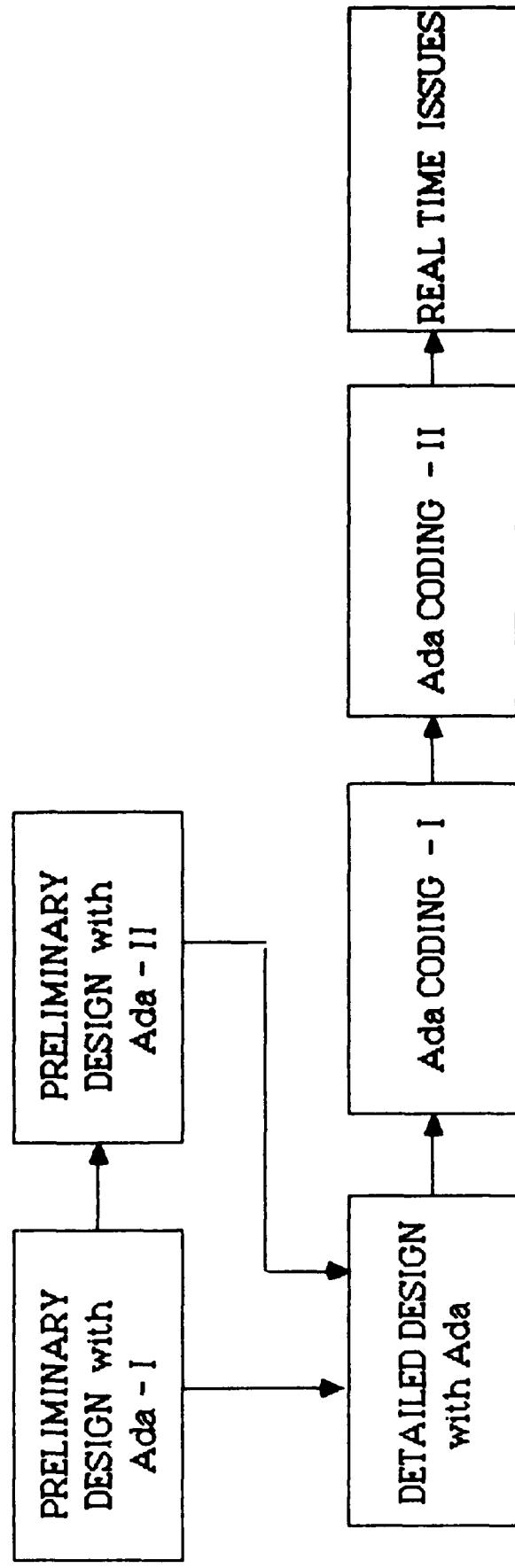
Figure 1. An Ada Training Series

26

2. BACKGROUND - Continued

Ironically the Coding Sequence had already been implemented and
test taught one time (at General Dynamics request). One
exercise in the Coding class is for small teams to implement an
adventure game [1]. One of the implementations of the adventure
game was extremely poor, violating all the principles of
software engineering.

That design, along with 2 others that were somewhat better, were
included in the Preliminary Design course, along with a design
created by TEXEL & COMPANY. At the appropriate point in the
Preliminary Design course the class is divided into groups of 3
or 4 students. Each group is provided with a statement of
requirements and asked to evaluate the first design against the
principles of software engineering. The groups work
independently and are instructed to return in 1 hour. Each
group then presents their findings to the other groups. The
sequence is repeated for each of the remaining 3 designs. The
results are astounding.

3. STATEMENT OF REQUIREMENTS

The statement of requirements that is distributed to the class
is shown in Figure 2. A simple adventure game is to be
designed. The goal of the exercise is not to develop the most
intricate adventure game, but to focus on the designs and their
adherence, or lack of adherence, to the principles of software
engineering. Bells and whistles can be easily added at a later
date if a "good" design exists for a simple adventure game.

4. DESIGN 1

The following paragraphs contain the elements of Design 1, the
first (and the worst) of the 4 designs in the Preliminary Design
class that the students evaluate. This is also the first design
that is distributed in the management seminar for evaluation.

The design consists of one package, Explore_Specs, with the main
routine, Explore_Driver, importing Explore_Specs.

STATEMENT OF REQUIREMENTS

ADVENTURE GAME

We are to design an adventure game.

Rooms:

   The game shall consist of a minimum of five rooms that are
   connected to each other to form a maze. Each wall may have
   an exit to another room (or be blocked) and the game
   consists of levels. A room may have an exit to a room in
   an upper level or lower level, for a total of 6 possible
   exits per room.

   Points are associated with each room. The adventurer
   accumulates these points when he enters the room for the
   first time.

   There is a message associated with each room. When the
   adventurer enters the room for the first time the message
   is displayed.

Treasures:

   Each room may or may not have treasures that are associated
   with the room. There must be a minimum of 4 treasures in
   the game.

   Each treasure is worth a certain number of points. As the
   adventurer roams through the maze, he accumulates points
   for each treasure acquired the first time the treasure is
   acquired. The adventurer's point total is decremented if
   the adventurer drops a treasure.

Commands:

   The adventurer may drop and pick up items as he travels
   through the maze. The adventurer may move from room to
   room. The adventurer may want to display the contents of a
   room and/or display current possessions.

Points:

   The player to accumulate the most points wins the game or
   the player that picks up a predetermined prize wins the
   game.

Figure 2. Statement of Requirements

28

4.1 Package Specification

```
package Explore_Specs is
     type Wall_Enum is (n, s, e, w, u, d);
     type Exit_Type is (None, Open, Closed, Locked);
     type Item_Enum is (Weapons, Treasures, Keys, Misc);
     type Backpack_Item_Enum_Type is array (1..10)
          of Item_Enum;
     type Action_Type is (Take, Drop, Throw, None);
     subtype Name_Subtype is String (1..10);
     subtype Desc_Subtype is String (1..160);
     type Item_Action is array (1..5) of Action_Type;
     type Monster_Type is (Norm, Gary, Parker);
     type Backpack_Type is array (1..10) of Name_Subtype;
     type Danger_Type is array (1..2) of Name_Subtype;
     type Person is
     record
          Backpack : Backpack_Type;
          Backpack_Item_Enum : Backpack_Item_Enum_Type;
          Item_Count : Integer := 0;
     end record;

     type Wall_Type is
     record
          Wall_Exit : Exit_Type;
          Next_Room : Integer;
     end record;

     type Item_Desc is
     record
          Name    : Name_Subtype;
          Action  : Item_Action;
          Lookup  : Integer;
     end record;

     type Item_Type is array (1..3) of Item_Desc;
     type Room_Wall_Type is array (Wall_Enum) of Wall_Type;
     type Room_Item_Type is array (Item_Enum) of Item_Type;

     type Room_Type is
     record
          Wall : Room_Wall_Type;
          Item : Room_Item_Type;
          Description : Desc_Subtype;
          Dangers : Danger_Type;
     end record;

     Player : Person;
     Room   : array (1..25) of Room_Type;
```

4.1 Package Specification - Continued

```
--************subprogram declarations*******

        function Length (Sentence : in String)
                return Integer;
        procedure Explore_Intro;
        procedure Open_Door (Room_Index :Integer);
        procedure Unlock_Door (Room_Index : Integer);
        procedure Check_Move (Room_Index : in Integer;
                Direction : in Wall_Enum;
                Ok        : out Boolean);
        procedure Describe (Room_Index : Integer);
        procedure Explore_Init;
        procedure Take_Item (Room_Index : Integer);
        procedure Drop_Item (Room_Index : Integer);

    end Explore_Specs;
```

4.2 Package Body

```
    with Text_IO; use Text_IO;
    package body Explore_Specs is

        function Length (Sentence : in String)
                return Integer is separate;
        procedure Explore_Intro is separate;
        procedure Open_Door (Room_Index :Integer)
                is separate;
        procedure Unlock_Door (Room_Index : Integer)
                is separate;
        procedure Check_Move (Room_Index : in Integer
                Direction : in Wall_Enum;
                OK        : out Boolean)
                is separate;
        procedure Describe (Room_Index : Integer)
                is separate;
        procedure Explore_Init is separate;
        procedure Take_Item (Room_Index : Integer)
                is separate;
        procedure Drop_Item (Room_Index : Integer)
                is separate;

    end Explore_Specs;
```

4.3   Main Routine

```
with Text_IO; use Text_IO;
with Explore_Specs; use Explore_Specs;
procedure Explore_Driver is

   -- type for commands


begin -- Explore_Driver

   -- the game


end Explore_Driver;
```

5.   EVALUATION OF DESIGN 1

Because it is impossible to replicate the entire class
discussion in a paper, only the high points are summarized
here.

5.1   Abstraction

Design 1 does not support abstraction very well. For example:

- o one package supplies all game information,
  there are no levels of abstraction
  - all types are visible to the main routine
  - all subprograms are visible to the main
    routine

- o poor choice of identifiers
  - wall_enum instead of Direction_Type,
  - n,s,e,w,u,d instead of north, south, etc

- o no user defined exceptions to represent error
  conditions that could be encountered by
  Take_Item, Drop_Item, and so on

- o Room_Index of type Integer is passed instead
  of a room

- o all boundaries are hard coded
  - Room : array (1..25) of Room_Type;
  - type Danger_Type...(1..2) of Name_SubType
  - type Backpack_Type...(1..10) of Name_SubType

5.2 Information Hiding

Design 1 does not support information hiding very well either. Yes, the subprograms are declared in the package specification and their implementation is deferred until the body. However all types and subprograms are visible to the main routine Explore_Driver. Finally both the player (Player : Person;) and the game board (Room : array (1..25) of Room_Type;) are contained in the Explore_Specs specification and are therefore visible to the user of the package, thus subject to modification.

5.3 Modularity

What is clear is that the main package is far too large. Goodenough defines modularity as "..purposeful structuring"[5]. All groups agree that Design 1 does not exhibit "purposeful structuring".

According to Constantine's [2] and Meyers [3] criteria for evaluating design by examining modules and their relationships, a package is not, and nor will it ever be, a module [4], - neither are subprograms declared locally within a declarative region (for example nested subprograms, subprograms declared within a package body, and so on) [4].

One can examine each subprogram and determine its level of coupling and cohesion but it is clear that the standard criteria need to be re-examined with respect to Ada designs [4].

5.4 Completeness

Design 1 is not complete because all components of the abstraction are not present. Design 1 does not provide a mechanism to accumulate points.

Design 1 does not satisfy the statement of requirements.

5.5 Confirmability

All groups agree that they simply would not know where to begin to test this design. All groups state that they would not let this design pass a Preliminary Design Review.

5.6 Localization

All groups report that logically related resources are collected in the one package. All groups also report that they "feel" that localization could be better but they are not sure how.

5.6 Localization - Continued

Most groups report that the command type and the subprograms
that operate on a command (getting and executing the command)
do not belong with the main routine but the groups are not
quite sure what to do with these command resources.

5.7 Uniformity

There simply are too many inconsistencies in Design 1 that, in
addition to the lack of abstraction, lead to confusion. For
example inconsistencies exist in the following areas: choice
of names, parameter modes, and subprogram selection. Each of
these is discussed below.

5.7.1 Choice of Names

Type names range from those that are meaningful (Exit_Type) to
a name like Item_Enum. What is Item_Enum?

Type names range from those suffixed with _Type (Exit_Type) to
those not suffixed with _Type (Person). Either suffix all
types with _Type or do not suffix all types with _Type.

Abbreviations are not consistent. The use of "n,s,e,w,u, and
d" for directions instead of "north south, east", and so on is
not consistent with use of Desc in Desc_SubType or Misc in
Item_Enum.

5.7.2 Parameter Modes

The parameter mode _in_ appears explicitly in one procedure
declaration (Check_Move) and is omitted in the rest, thereby
chosen by default.

5.7.3 Subprogram Selection

There are only two subprograms that return a value: Length
and Check_Move. One is a function and the other a procedure.
Why?

5.8 SUMMARY

All groups agree that Design 1 does not support the principles
of software engineering. A complete redesign is required.

6. DESIGN 2

The following design is the last (and the best) of the 4
designs that the Preliminary Design class evaluates. This
design is the second (and last) design that the management
class evaluates.

6.1 Command_Package Specification

```
package Command_Package is

    type Command_Type is private;

    procedure Get     (Command : out Command_Type);
    procedure Execute (Command : in Command_Type);
    function Done return Boolean;

    Bad_Command : exception;

private

    type Command_Type is access String;

end Command_Package;
```

6.2 Main Routine

```
with Command_Package; use Command_Package;
procedure Play_the_Game is

    Command : Command_Type;

begin -- Play_the_Game

    COMMAND_LOOP:
    loop
        Get (Command);
    exit when Done;
        begin
          Execute (Command);
        exception
          when Bad_Command =>
              Put_Line "("Invalid command");
              Put_Line "("Enter another");
        end;
     end loop COMMAND_LOOP;

end Play_the_Game;
```

6.3   Command_Package Body

```
with Text_IO; use Text_IO;
with Game_Package; use Game_Package;
package body Command_Package is

    -- Execute parses the command
    procedure Parse (Command : in Command_Type;
                         Verb : out Verb_SubType;
                         Noun : out Noun_SubType)
               is separate;

    procedure Get (Command : out Command_Type)
               is separate;
    procedure Execute (Command : in Command_Type)
               is separate;
    function Done return Boolean is separate;

end Command_Package;
```

6.3.1   Game_Package Specification

```
package Game_Package is

    type Room_Names_Type is
         (Dungeon, Banquet_Hall, Kitchen,
          Throne_Room, Bedroom, Bathroom, None);
    type Words_Type is
         (move, pick_up, drop, display, stop,
          gold, diamonds, silver, Ada, game,
          room, my_status, north, east, south,
          west, up, down);
    subtype Verb_SubType is Words_Type
         range move .. stop;
    subtype Noun_SubType is Words_Type
         range gold .. down;
    subtype Treasures_SubType is Noun_SubType
         range gold .. Ada;
    subtype Directions_SubType is Noun_SubType
         range north .. down;

    procedure Move    (Where  : in Directions_SubType);
    procedure Pick_Up (Object : in Treasures_SubType);
    procedure Drop    (Object : in Treasures_SubType);
    procedure Display (Room   : in Room_Names_Type);
    procedure Display_Players_Status;

    No Exit   : exception; --raised by Move
    No Object : exception; --raised by Pick_Up or Drop

end Game_Package;
```

6.3.2 Game_Package Body

```ada
with Text_IO; use Text_IO;
package body Game_Package is

    type Exits_Type is array
        (Directions_SubType) of Room_Names_Type;

    type Treasures_Info_Type is
    record
        First_Time : Boolean := False;
        Points     : Positive;
    end record;
    Game_Treasures : Treasures_Info_Type;
    type Treasures_Set_Type is array
        (Treasures_SubType) of Treasures_Set_Type;

    Max_Points_per_Room : constant Postive := 50;
    type Rooms_Type is
    record
        Exits     : Exits_Type;
        Treasures : Treasures_Type;
        Points    : Positive range 1..Max_Points_per_Room;
        Message   : String (1..40);
    end record;

    type The_Game_Type is array
        (Room_Names_Type) of Rooms_Type;
    The_Game : The_Game_Type;

    Max_Points_per_Game : constant Positive := 500;

    type Player_Type is
    record
        Points    : Positive range 1..Max_Points_per_Game;
        Location  : Room_Names_Type;
        Treasures : Treasures_Set_Type;
    end record;
    Player : Player_Type;

    procedure Initialize_Game is separate;
    procedure Display_Initial_Greeting is separate;

    procedure Move     (Where : in Directions_SubType)
                is separate;
    procedure Pick_Up (Object : in Treasures_SubType)
                is separate;
    procedure Drop     (Object : in Treasures_SubType)
                is separate;
    procedure Display (Room : in Room_Names_Type)
                is separate;
    procedure Display_Players_Status is separate;
begin -- Game_Package
    Initialize_Game;
    Display_Initial_Greeting;
end Game_Package;
```

36

7. EVALUATION OF DESIGN 2

7.1 Abstraction

At the highest level of abstraction, an adventure game
consists of a player who sits at a terminal and repeatedly
enters commands and waits for the command to be executed. The
player quits when he is through or when he has achieved some
goal. At this level of abstraction knowledge of the legal
words of the game is not required.

At the next level of abstraction, we have a player who moves
from room to room picking up and dropping objects until some
predetermined prize is found or the player quits. The player
can optionally display a room contents or his own possessions.
At this level the legal words of the game and the legal
operations of the game must be defined so that Execute may
call the appropriate routine to execute a command. Note that
although the legal words of the game must now be declared, at
this level the game board and point scheme do not need to be
defined.

At the final level of abstraction, the maze, player and point
scheme must be defined.

Note that these levels are supported by the Command_Package,
Game_Package specification and Game_Package body respectively.
Design 2 exhibits good use of abstraction. The main routine
only has access to the Command_Package. The Command_Package
only has access to the Game_Package specification. Finally
the Game_Package body contains the game and the player.
Details are deferred until the last possible moment.

Note also that the remaining deficiencies of Design 1 have
been removed.

7.2 Information Hiding

Abstraction helps decide what to hide [5]. Because of proper
abstraction in this design, the details that are suppressed at
one level are hidden. For example the game itself is hidden
from the main routine; only the command package is visible to
the main routine. As another example of good information
hiding, the game (Game : Game_Type;) and the player (Player :
Player_Type;) are declared in the Game_Package body and are
only accessible to the implementation of the routines that
repeatedly update the game and/or player status.

Design 2 exhibits good information hiding.

7.3 Modularity

Although a package is not a module in the classic sense of the word (see Section 5.3), commands are contained in a command "module" and game information is contained in a game "module".

The subprograms provided are cleaner than in Design 1.

7.4 Completeness

All components of the abstraction are present (Is this true?). A point system now exists. The design maps to the statement of requirements.

Design 2 exhibits completeness.

7.5 Confirmability

At the highest level the command package can be tested. At the second level the definition of the game can be tested. At the third level the game itself can be tested.

Design 2 is confirmable in logical steps. Each level of abstraction is readily testable.

7.6 Localization

The command and its logical operations are packaged in one package, Command_Package. The description of the game is contained in the Game_Package specification while the game itself is implemented in the Game_Package body. Code that is logically related is physically co-located.

Design 2 exhibits localization. The groups now see how to implement what they intuitively felt was required, a better localization of resources. The class learns that what was needed was a better abstraction.

7.7 Uniformity

All the inconsistencies of Design 1 are gone. Design 2 exhibits uniformity because:

> o naming conventions are employed
>
> o abbreviations are not present
>
> o parameter modes are explicitly stated for all procedures, not for functions because functions can have parameters of mode in only

8.0 DEPENDENCY DIAGRAMS

As a final step, the class is shown the dependency diagrams shown in Figure 3. A visual inspection of the diagrams for the two designs clearly shows how the Game and the Player are visible to the world in Design 1 (declared in Explore_Specs specification) and how they have been suppressed to the third level in Design 2 (declared in Game_Package body).

9.0 APPLICABILITY TO MANAGEMENT

The applicability of the pedagogical approach for software engineers has been successfully demonstrated in this paper. Because of its usefulness the exercise was refined for use in the three (3) day Ada for Technical Management course. The last 2 hours of the last day of class are devoted to comparing 2 of the 4 designs evaluated in the Preliminary Design course, specifically the worst and the best (the 2 designs contained in this paper). The results are extremely encouraging. Not only do the managers really begin to understand software engineering, but they begin to appreciate why education in software engineering with Ada requires more time than training a programmer in FORTRAN.

10.0 CONCLUSION

There are 2 design flaws in Design 2. Each group of individuals (software engineers and managers) has been able to detect the 2 design flaws. Their ability to read and comment effectively on Ada designs, prior to having a coding class, is proof that one does not need to know everything there is to know about Ada in order to effectively participate in design reviews.

In this author's opinion design comparison yields positive results, for both managers and software engineers. Inclusion of this technique in future Ada related courses can only enhance the students' learning process.

DESIGN 1
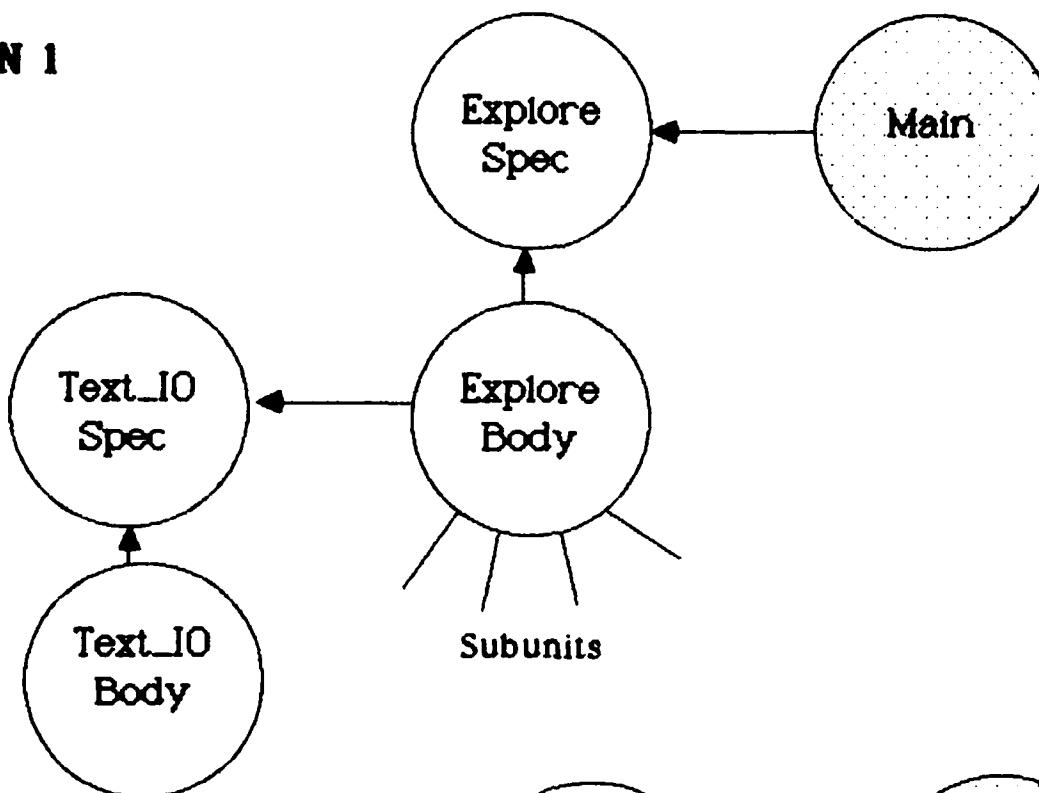
DESIGN 2
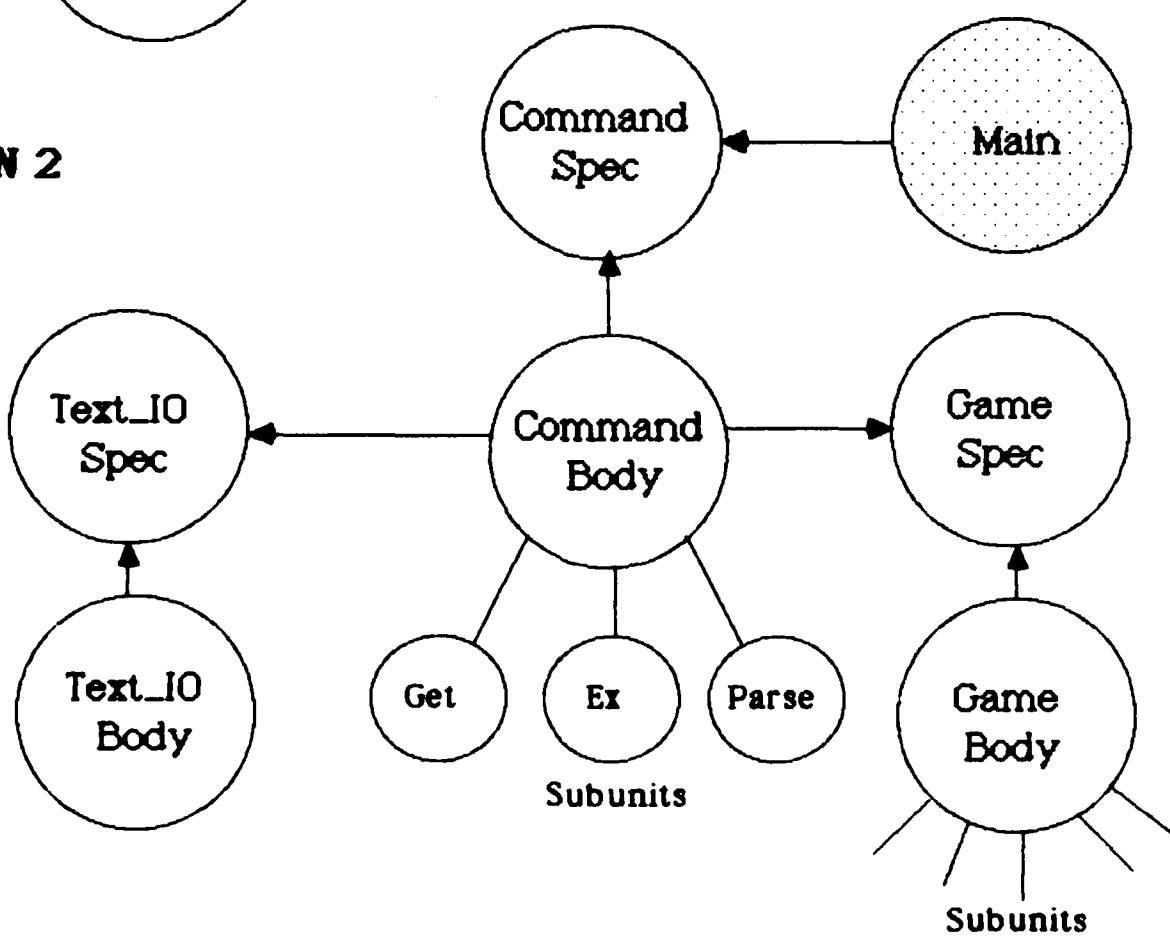
Figure 3. Dependency Diagrams

Bibliography

[1] The adventure game as an exercise was originally
conceived by D. Bolz (USAF Ret.)

[2] Yourdan, E. and Constantine, L., <u>Structured Design</u>, 2nd
Edition, Yourdan Press, NY, NY, 1978.

[3] Myers, G., <u>Composite Structured Design</u>, VanNostrand
Reinhold, NY, NY, 1978.

[4] Hammons, C. and Dobbs, P., "Coupling, Cohesion, and
Package Unity in Ada", AdaLETTERS, Vol IV, No 6, May/June
1985.

[5] Ross, D., Goodenough, J., and Irvine, C., "Software
Engineering:Process, Principle, and Goals", COMPUTER Magazine,
May 1975.

This page left blank intentionally

Instantiating Ada at the University of Central Florida (and Why There Are Few Good Texts on Ada and Software Engineering)

by

Darrell G. Linton, PhD, PE
Associate Professor
Computer Engineering Department (CEBA 207)
University of Central Florida
Orlando, Florida 32816
(305)275-2236

Abstract

This paper describes how Ada software engineering education is
implemented in the Computer Engineering Department at the
University of Central Florida. In addition to the format of the
courses, the textbooks chosen and the available hardware and
software, the need for more complete texts and more efficient
software are addressed. Failure to correct these deficiencies has
the potential of delaying the acceptance of Ada and reducing the
number and quality of software engineering graduates.

Introduction

During the academic years 1985-86 and 1986-87, Software Engineering
with Ada has been taught as a two-course sequence in the Department
of Computer Engineering at the University of Central Florida. The
first course (ECM5806, Software Engineering I) covers the Ada
language (using Saib, Ada: An Introduction) and the principles of
software engineering (using Fairley, Software Engineering Concepts)
and the second (ECM6807, Software Engineering II) is a project
course where teams of two to four students design, develop,
implement and document a system in Ada. Both courses are graduate
level although undergraduates may receive credit for Software
Engineering I.

The ECM5806 class is taught live on the Orlando campus but is also
taped and sent to several other locations throughout the state of
Florida. This provides students who work full-time with the
opportunity to persue a Master's degree without the necessity of
driving to Orlando for classes. Exams and homework are transferred
via a courier service and off-campus students are about two days
(one or two lectures) behind the live class. The ECM6807 course is
not taped (since it is a project class) and meets late in the
afternoon, twice per week, so that working students can attend.
Class time is spent on implementation problems experienced by each
team and periodic status updates (including execution of the
current system in front of the class).

Hardware and Software

Students in the College of Engineering (COE) have access to two

mainframes, an IBM4381 (with the VM/CMS operating system) and a
Gould 32/6780 (with UTX, Gould's version of the UNIX operating
system), both of which run a validated version of the TeleSoft Ada
compiler. Although students often choose to use company-owned Ada
facilities, approximately 60% of the class use university
equipment. Turn-around time (i.e., the time required to compile and
execute a main procedure) on the Gould is about 30 seconds to 1.5
minutes while on the IBM, it is on the order of 6 to 10 minutes.
Part of the reason for this discrepancy is that the IBM services
the entire university community while the Gould is used primarily
by COE students (most of which are in the Ada course). As of this
writing, there are five hardwired terminals (located in the COE and
available 8am - 5pm) and four modem lines into the Gould, and 100+
terminals (located on campus and throughout central Florida and
available 24 hrs per day) and fifteen modem lines into the IBM.
Thus, although the Gould has a smaller turn-around time, the IBM is
more accessible. In any case, turn-around time is poor compared
with what most students (and practicing engineers) are used to and
a fast (efficient), microcomputer-based, reasonably priced
(including the cost of a site license), validated Ada compiler is
sorely needed by the university community. (An Alsys validated Ada
compiler for the IBM PC/AT is on order at the COE but it is
relatively expensive.)

The 'Best' Ada Text

Although it was not surprising, it was discouraging to find not one
single text which contained the minimum features an instructor
should expect. For instance, if key terms like 'instantiate',
'files' and 'context clause' are not in the index of a text or,
although present in the index, none of the page references provide
an understandable English definition of the term, the book is not
appropriate for classroom use. The lack of <u>complete</u> examples (the
shorter, the better) is another deficiency of numerous Ada texts.
Many authors simply do not understand that someone learning any
computer language must be provided with at least four things:

   a. complete source listings (preferably, computer printouts
   from working source files) of all compilation units used in
   each example,

   b. a keystroke-by-keystroke description of exactly what the
   user entered when the program was executed,

   c. a complete listing of the output produced by the program,
   and

   d. a discussion of a., b. and c.

It would be especially nice if, perhaps in appendices, commands for
compiling, editing, linking and executing Ada programs which run
under UNIX, CMS and VMS were shown and illustrated with example
programs. Textbook writers often forget that there are at least two

languages that must be learned when studying a computer language —
the language itself and the commands associated with the operating
system. Most computer manuals which discuss, for example, how to
perform separate compilations using an Ada compiler and linker, can
be understood only by experienced systems programmers (who are also
familiar with Ada).

What, then, is the best Ada text? There does not exist such a thing
in this author's opinion. The most complete and understandable
handbook on Ada is by Cohen, Ada as a Second Language — this book
is a must for anyone who is serious about Ada. Texts with many
examples (some of which are incomplete) include: Saib's Ada: An
Introduction, Wiener and Sincovec's Programming in Ada, Price's
Introduction to Ada and Vasilescu's Ada Programming with
Applications. One of the few texts which details the many
subprograms from packages text_io, sequential_io and direct_io is
by Elbert, Embedded Programming in Ada. A text specifically on
parallel programming is Cherry's Parallel Programming in ANSI
Standard Ada.

What is the 'best' Software Engineering Text?

The Fairley text mentioned above is the 'best' software engineering
text that the author is familiar with. However, the real problem is
not the availability of texts —it is how to teach software
engineering in a classroom environment and how to grade the
assignments. Topics such as software requirements definition,
software design, implementation, validation techniques and
maintenance can all be discussed in the classroom but testing the
understanding of these principles requires a term project. Grading
a project, however, often becomes a series of subjective judgements
based in part on writing style and the use of correct English. In
fact, it is often the use of English that distinguishes the quality
of one project from another.

Few software engineering texts provide examples or exercises which
could be given on an in-class exam. In addition, most topics can be
read about and understood without in-depth classroom discussions.
Case studies which illustrate good and bad applications of software
engineering principles and which are discussed with respect to a
standard (e.g., DOD-STD 2167) are not presented in today's texts.
Until and unless the case study concept is incorporated into
software engineering texts, it will be difficult for academia to
produce high quality software engineers. Thus, as of this writing,
the author is unaware of any software engineering texts which meet
the needs of the engineering community.

Conclusions

The primary goals of this paper are to: 1. share what is happening
in Ada software engineering education and training at UCF, 2.
indicate the need for fast and inexpensive Ada compilers for
classroom use and 3. point out deficiencies in the available

textbooks on Ada and software engineering. Goals 2 and 3 are intended to encourage all those interested in the future of Ada software engineering education to improve the existing compilers and fill in the gaps left by many of today's texts. Failure to do so, in this author's opinion, will reduce the number and quality of software engineering graduates as well as delay the acceptance of Ada by the engineering profession.

References

Booch, G., Software Engineering with Ada (2nd Ed.) (Benjamin/ Cummings,Menlo Park, CA, 1986).

Cherry, G., Parallel Programming in ANSI Standard Ada (Reston Publ., Reston, VA, a Prentice-Hall Co., 1984).

Cohen, N., Ada as a Second Language (McGraw-Hill, New York, NY, 1986).

Elbert, T., Embedded Programming in Ada (Van Nostrand Reinhold, New York, NY, 1986).

Fairley, R., Software Engineering Concepts (McGraw-Hill, New York, NY, 1985).

Price, D., Introduction to Ada (Prentice-Hall, Englewood Cliffs, NJ, 1984).

Saib, S., Ada: An Introduction (Holt, Rinehart and Winston, New York, NY, 1985).

Vasilescu, E., Ada Programming with Applications (Allyn and Bacon, Boston, MA, 1987).

Wiener, R. and R. Sincovec, Programming in Ada (John Wiley and Sons, New York, NY, 1983).

Treatment of the Ada Language in a
Programming Language (ACM-CS8) Course

Michael R. Meeker
Department of Computer Science
University of Wisconsin — Oshkosh

The 1978 ACM recommendations for undergraduate Computer Science curricula make provision for a course dealing with the organization of programming languages. It is recommended that this course be required of all students majoring in the area. Typically, students would have completed two courses in computer programming, including treatment of structured programming, debugging, data structures, and recursion. In addition, it is highly recommended that students complete courses in assemblers and file processing before enrolling in the course. The objectives [1] of the course are:

"(a) to develop an understanding of the organization of programming languages, especially the run-time behavior of programs;

(b) to introduce the formal study of programming language specification and analysis;

(c) to continue the development of problem solution and programming skills introduced in the elementary level material."

In order to accomplish these objectives it is necessary to develop the context for classification and description of programming languages in general, as well as to illustrate aspects of programming languages by citing their implementations in various languages. To that end, Ada is an attractive language to study because it possesses many important features of which intermediate level students are unaware. However, Ada is a large language and therefore much detail must be taught before the student is in a position to do programming exercises that illustrate its advanced features. One purpose of this paper is to identify those topics within a programming languages course where Ada provides outstanding examples.

Students at this level have only limited knowledge of the principles of software engineering. It is difficult for them to appreciate the language design decisions that are so fundamental to Ada. Unless some design justification accompanies treatment of the language features, students are likely to develop an indifferent attitude toward many of Ada's strong points. The second purpose of this paper is to relate some of the software engineering concepts important to student appreciation of Ada's design.

EXPLOITATION OF ADA'S STRONG TYPING

By the time computer science students reach the intermediate level, there is a good possibility that they have encountered a language with some data typing and enforcement features. Most commonly this experience comes through a knowledge of the programming language Pascal. Regardless of their previous language experience, it is unlikely that these students have used typing to its full advantage. One of Ada's prominent features is the ability to specify derived data types. The use of derived data types becomes the basis of strict enforcement in Ada. Since the derived data type is not a feature in many languages (Pascal in particular), it is necessary to familiarize the student with the techniques as well as the justifications for derived data types. In order to take advantage of strong type checking, students must learn to

recognize circumstances in which typing errors could be prevented by using derived types. Some straightforward examples are:

- real abstractions are measured in different ways (scales), e.g. the abstraction temperature could be represented in Fahrenheit, Celsius, or Kelvin.
- integer abstractions are used to count different objects, e.g. 5 apples + 22 autos => ????.
- separate enumerations with equivalent functionality, e.g. ASCII and EBCDIC character sets.

Most Ada textbooks cover data type features, but unfortunately do not provide clear discussions of how effectively to use strong type checking. While this is probably acceptable for experienced programmers, it is insufficient for intermediate level students.

ADA ABSTRACTIONS

Most intermediate computer science students' experience with software development is with relatively simple software systems. These systems are usually simple enough that an individual programmer can be responsible for the development of the complete system. Usually the software system has a brief life cycle, ending when the assigned project is submitted for instructor evaluation. Hence, the issues of information hiding, need to know, localization, and maintainability hardly, if ever, surface.

On the other hand, most computer science students have dealt with numerous abstractions without being consciously aware that they were doing so. Some commonly used abstractions are data and program files, compilers, link editors, and various operating system services. In introductory programming courses, the uses of abstractions are primarily motivated by concerns of understandability (data abstraction) and structured programming concepts (algorithmic abstraction). At the introductory level, these concerns are only loosely related. An important lesson to be learned by Ada students is that data and algorithmic abstraction go hand in hand[5].

One approach to the effective use of abstraction is to envision a data abstraction hierarchy [2]. At the bottom of the hierarchy are data objects associated with implementation detail, at the top are data objects of great abstraction. Once such a hierarchy is established it is possible to deal with algorithms at each level. This hierarchical approach has the advantage of making apparent the information hiding and need to know issues. It logically separates the implementation aspects of the abstraction from its use. It is at this point that packaging logically fits into the course outline.

FAULT TOLERANT ADA SYSTEMS

Most beginning computing assignments are designed such that the input data when properly processed will yield correct results. Seldom do we introduce the complexity of error conditions into our examples. Exception conditions are therefore manifest as programming faults which must be eliminated by modification or correction of mainline code. This oversimplified perception of programming systems needs to be modified at the intermediate level in the curriculum.

It is often the case that the languages which are familiar to students provide

little or no exception handling capabilities. This makes the design of fault tolerant programs difficult or impossible to specify. Ada provides fault tolerant mechanisms in a variety of ways:

- standard exceptions are always present.
- data types may be constrained.
- the programmer may pretest domains with the "in" and "not in" operators.
- the programmer may control the exception handling with Ada code.
- the exception may be propagated to a higher level by deferring exception handling or re-raising the exception.
- provision is made for premature exit from iterative processes.
- the programmer may create user defined exceptions.

The programming techniques that produce fault tolerant systems are straightforward. They do, however, require that the students expand their comprehension of sequence control mechanisms. As a result of the wide acceptance of structured programming techniques, most students have been indoctrinated with the principle that program statements in textual sequence correspond to the execution sequence [4]. This principle must be modified to accommodate exception handling. A second issue of interest is the propagation and/or re-raising of exceptions so as to refer the exception to a higher level of abstraction. The questions of where as well as how to handle an exception are new topics for intermediate level students.

TASKING AND CONCURRENCY IN ADA

There is little doubt that computer technology is heading in the direction of parallel processing. At the present time we are in the precarious position of having both feet firmly planted on single processor software technology ground that is soon to erode. Little is being done to teach or even introduce undergraduate students to parallel programming techniques. In the opinion of the author, teaching parallel programming techniques should be a high priority task within the computer science curriculum.

At the present time many computer science programs do not have access to multiprocessor systems. However, Ada provides the appearance of such[3]. This is sufficient to begin student orientation to parallel systems.

CONCLUSIONS

The incorporation of a discussion of Ada into the intermediate programming language (ACM-CS8) course is justified. Ada contains many language features that are not commonly taught or emphasized in prerequisite courses. Ada is designed with consideration for software engineering principles. Therefore, a discussion of Ada features serves to orient the intermediate student to this topic. Finally, Ada incorporates language features that are compatible with the current direction of technological development. Ada represents a programming tool for the present as well as the future. Part of the definition of a well rounded computer science student should include knowledge of the Ada programming language. The programming language course provides a convenient and relevant context to have this happen.

REFERENCES

[1] Austing, R.H., Barnes, B.H., Bonnette, D.T., Engel, G.L. and Stokes,G. Curriculum'78: Recommendations for the Undergraduate Program in Computer

Science. Comm. ACM 22, 3, 147-166.

[2] Booch, G. [1987] Software Engineering with Ada. 2nd Ed. Benjamin/Cummings, Menlo Park, CA.

[3] MacLennan, B. [1987] Principles of Programming Languages: Design, Evaluation, and Implementation. Holt, Reinehardt and Winston, New York, NY.

[4] Pratt, T.W. [1984] Programming Languages. Design and Implementation. 2nd Ed. Prentice-Hall, Englewood Cliffs, NJ.

[5] Wirth, N. [1986] Algorithms and Data Structures. Prentice-Hall, Englewood Cliffs, NJ.

Ada from the Trenches : A Classroom Experience

Jaime Niño
Computer Science Department
University of New Orleans

Introduction.

In August of 1984, the department of Computer Science of the University of New Orleans started implementing the programming language Ada as the departamental language. After almost 3 years into the project, we feel we have succeeded. The purpose of this paper is to share our experience to the Computer Science educators and Ada users at large.

Structure of paper.

The paper is composed of the following topics :
University Setting.
Departamental Setting
Curriculum overview.
Ada as a Primary Programming Language.
Ada curriculum implementation history.
Ada teaching experience.
Student Population Response.
Compiler Experience.
Book review.
Conclusion.

University Setting.

The University of New Orleans (UNO) is an urban university of approximately 17,000 students situated on Lake Ponchatrain in New Orleans, La. It is an urban public university, the second largest of the institutions governed by the Louisiana State Board of Regents.

The UNO academic computer resources consists primarily of a cluster of four VAX 8600 16 Mb computers running VMS Version 4.5. and DECNET and approximately 150 Zenith microcomputers with Winchester discs and numerous terminals throughout campus. There is also a campus-wide Ethernet. Computer Science undergraduates taking core courses are supported by this system.

Departament Setting.

We offer the baccalaureate in Computer Science and in conjuction with the Mathematics Department we offer a master degree in Mathematics with specialization in Computer Science. The department consists of 10 full time faculty members all of which are expected to teach programming classes. We are supported by part time faculty, several teaching assistants and paper graders.

There are approximately 400 Computer Science majors at UNO, and typically 1500 students studying Computer Science courses at any one time. In any given semester, there are approximately 120 declared Computer Science majors studying CSCI 1583, the first Computer Science course for our majors, comparable to CS1 in the terminology of the ACM Curriculum Committee [Augt 79]. There are approximately 90 students enrolled

in CSCI-2120 (or CS 2 in the ACM curriculum) and 60 in CSCI-2125 (Data Structures or CS 7).

UNO is on a semester system, with most Computer Science courses having three hours of lecture per week, and offering three credits to the student.

Curriculum Overview.

The University of New Orleans Computer Science Curriculum [UNO 84] closely resembles the ACM Curriculum '78. For this paper we will take a closer look at the syllabus of our introductory courses CSCI-1583, CSCI-2120 and the Data Structures course CSCI-2125, for which programming is a major component.

CSCI-1583
Prerequisites : Plane trigonometry with Algebra
Corequisites : Calculus I or Discrete Mathematics. We advise the students to coregister in Discrete Mathematics.

Syllabus overview:
The major topics of study in this course are computers in general programming principles and the Ada language.

Computers in general covers computer systems organization, basic computer organization and history of computers.

Programming priciples covers programming languages concepts, examples of programming languages, typical programming tasks, software lifecycle, software quality, algorithm design (top-down, bottom-up) with strong concentration on top-down design and step-wise refinement. Structure Programming, Abstract data types, algorithm testing, documentation.

From Ada we cover primitive types, data manipulation via typed objects, control statements, subprograms, scope and visibility, records with fixed types, arrays, and use of packages. Attributes and Input/Output using the standard input and output files.

Programming assignments size varies from small (tens of lines) to moderate size (no more than 1000 lines).

CSCI-2120
Prerequistes : CSCI-1583 and either 1) credit in Discrete Mathematics or 2) concurrent registration in Discrete Mathematics and credit in Calculus I

Syllabus overview.
Introduction : Software life cycle. Ada Review Data types Types. Subtypes Simple types : Integer, Float. Boolean. Character Type classification Control Structures Subprograms. Structured Programming Structured Types Records and arrays More on data types. Abtract Data Types Encapsulation, localization and Information Hiding Design Techniques : Top Down design Bottom Up Design Separate compilation and top-down coding. Top-down testing. Other design techniques Ada Block structure Scope and visibility Recursion and backtracking Programming in the large Packages. Bottom up design and packages Robustness Exceptions Files Testing (Structured walk throughs) and verification Program assertions loop invariants Partial program correctness

The student is given on average 5 programs whose size varies from 500 lines to 1500. Students are also given a programming project whose size is between 2000 to 3000 lines.

CSCI-2125
Prerequisites : CSCI-2120 , Discrete Mathematics and Calculus I.

Syllabus overview.
Abstract Data Types : Specification, design, validation , implementation. Study of typical data structures as ADT's : Stacks, Queues, Lists, Recursive Types, Binary trees, General trees, Graphs. ADT's and algorithms : searching, sorting, hashing. Other ADTS's.
From Ada : Generic Packages. Access Types. Variant Records.

Ada as a primary programming language.

The term "primary programming language" is used in [Auge83] to describe a language taught in the initial courses of a Computer Science curriculum and thereafter used as a standard language of reference.

There has been a great deal of debate in the Computer Science community about primary programming languages. [Auge83] discusses the relative merits of Pascal, PL/I, and Ada as primary programming languages. The ACM Curriculum Committee [Koff84] has recommended that only Pascal, PL/I and Ada meet the criteria for an acceptable language in the revision of their earlier proposal for CS 1.

In [Evans et al 1985] the autors cite several reasons against the adoption of Ada as a primary programming language. Among the reasons cited were the complexity and the size of the language, the lack of compilers and textbooks. Of those reasons one may still argue against the adoption of Ada on the basis of its complexity. After three years of teaching Ada we have demonstrated that we can find a suitable subset of Ada that does not do a disservice to the language and that supports and serves us well in our teaching endeavors.

Many universities in North America have created an Ada course in the Computer Science curriculum [SigAda 87]. Almost no universities have adopted Ada as the primary programming language. We in the compute Science Department at **UNO** adopted Ada as the primary language in the fall of 84. To aid in the understanding of our choice of Ada as the departamental language I proceed to state our common teaching philosophy and goals, at least with respect to the programming courses.

In our teaching endeavors we teach and stress the principles and fundamentals of the art of programming. We underplay "programming tricks" in behalf of readable, maintainable and correct programs. Software engineering has developed well proven principles which aid in the development of software. Some of those principles are structure programming, modularization, information hiding, data abstraction, encapsulation and localization. Software engineering also provided techniques and guidelines for programming in the large. Our philosophy lies on the teaching of fundamentals of programming; one of our major goals on the teaching of programming is to teach the principles and the methodology necessary to produce a program that is correct, reliable, robust, maintainable, verifiable and portable. Therefore we teach the principles and techniques available from software engineering that aid us in attaining this goal.

Ada is a modern programming language which was designed to support all of these principles and the needs of modern software development. Using Ada we have a unique language were the students get the opportunity to put the principles. taught in class to practice The advanced features found in Ada to support the development of embedded

systems has given us the opportunity to be able to use Ada in the upper level courses which have a programming component. We can therefore say that Ada has unified our curriculum as we can use Ada throughout our curriculum as a programming tool.

In the core programming courses we use Ada as a tool to illustrate programming techniques and principles. The learning of Ada per se is not a goal in our programming courses. What this means is that we teach the necessary Ada syntax and semantics to illustrate programming principles and techniques. Ada is a very rich and complex language whose wealth of features can overwhelm any well experienced programmer. We had to carefully trim Ada to make it into a manageable language from the point of view of the teaching of programming. We can certainly say that students that go through our core programming courses adquire an excellent foundation in Ada; they can put Ada in use to produce high quality programs using the standards of software engineering; we cannot say that they are Ada experts; this is not one of our goal.

One can certainly argue that Ada is too complex a language to be taught in an introductory course in programming. My response to that argument is based on the intended goals for that course. If the goal is to teach Ada per se, I support that argument. If the goal is to teach programming using Ada as a tool, I do not see any extra burden put on the student by choosing Ada, I only see benefits from the choice. I am of the opinion that the difficulty of teaching introductory programming lies on the subject matter itself; programming is a difficult task, and this task can be ameliorated or worsen with the programming language chosen. If an introductory course is aimed to the teaching of programming as an art with well proven principles and techniques, Ada can provide a programming environment where theory can be put into practice. For a case in point, it is a lot easier to show in Ada how one can implement the information hiding priciple that say, in Pascal. Ada not only have subprogram, but packages and private types.

Ada curriculum implementation history.

We started the implementation of Ada in the fall of 84 with the course CSCI-1583. In the fall semester two sections were offered to a initial total of 60 students. We continue offering CSCI-1060 which is the equivalent course using Pascal.

In the spring of 85 we offered two sections of CSCI-1583 and one section of CSCI-2120 using Ada. We offered one section of CSCI-1060 and one section of CSCI-2120 using Pascal.

In the fall of 85 we offered CSCI-1583, CSCI-2120 and CSCI-2125 using Ada. We continued facing out the sections of CSCI-2120 and CSCI-2125 using Pascal that Computer Science major could take for credit.

By the Spring of 1986 the only section remaining to be faced out from our curriculum was a section of CSCI-2125 using Pascal.

In the fall of 1986 we taught all the programming core course sections of CSCI-1583, 2120, 2125 using Ada. There were no sections of any of those courses offered using Pascal in that semester or afterwards. We continue teaching an introductory course using Pascal which is aimed to the Liberal Art students as well as an introductory course using FORTRAN aimed to the Engineering School students.

By the fall of 1986 faculty teaching higher courses with a programming component could expect to have many of the students in the course with Ada experience. Not all upper level students can be assumed to have Ada experience at the time of this writing. But their number is rapidly diminishing.

Ada Teaching Experience.

I proceed to relate the experience per se of the teaching of Ada in the courses CSCI-1583, CSCI-2120, CSCI-2125.

The main goal in CSCI-1583 is the teaching of the elements of structured programming, data abstraction, algorithm development and the necessary syntax to support such activities.

In 1583 the students get exposed to the following concepts:
1. Structure programming.
2. Abstract Data Types.
3. Top-down design.

Ada serves well each of the concepts listed above. Ada has a complete and fully braketted control structure set. Ada distinguises between OUT and IN OUT parameters in contrast with Pascal and its by-reference parameter mode. Having given the definition of abstract data types, we can illustrate it via the Ada primitive types and the fact that no implicit coercion is allowed in Ada. We can also illustrate it by introducing derives types of primitive types. Students can be given the opportunity of using non-primitive types via packages provided by the instructor. With this simple instance, an instructor can illustrate Abstract Data Types, team programming as well as the information hiding principle. Top down design is supported in Ada via subprograms and separate compilation and this last Ada feature gives the instructure an oportunity to give the student the experience of programming in the large.

From the point of view of concept formation (programming in this case), students can be given a package or a set of packages developed by the instructor, and the students write simple drivers for those packages. The instructor can use separate compilation to give the opportunity to students of being part of the writing of a rather "interesting" program by assigning the students the writing of a subprogram which will be linked to a main subprogram written by the instructor. This feature can be exploited further by given different group of students different subprograms to write.

Notice that in these situations the students can be made part of a programming effort with a minimum knowledge of Ada syntax. To use packages students need to know how to declare and give value to the arguments how to use the conditional and while control structures, how to call subprograms and to include packages with their drivers. The actual writing of subprograms can be helped by giving simple by well defined operations to implement, whose logic is simple and do not need tne use of sofisticated Ada features.

The goal of the second course CSCI-2120 is to teach to the student the principles underlying the production of programs which are correct, portable, maintainable, modifiable and verifiable.

In the teaching of the second course the choice of Ada is more and more rewarding. Among the main concepts taught in this course we have:
1. Structure Programming revisited.
2. Design methodologies . (Top-down , bottom-up, among others).
3. Robust programming and error trapping.
4. Recursion and Backtracking.
5. Abstract Data Types revisited.

The concept of abstraction which was taught using subprograms can be reenforced using unconstrained arrays and discriminated records and user defined enumeration types. Using separate compilation student can get experience in top-down design . top-

down coding and top-dow testing. With packages students learn bottom-up design and coding Exceptions simplifies the introduction and the actual implementation of error trapping.

At the time the course CSCI-2125 is taken, the student has a good backgroun in programming in Ada. In this course the students get experience in the specification, design and implementation of Abstract Data Types. At this point packages, private and limited types will support the concepts of information hiding, encapsulation and data abstraction.

From the teaching point of view, one of the strengths of the programming language Ada is in the fact that it supports all the modern principles of software engineering, making the teaching of such an easier and rewarding task.

I am the first to admit that teaching Ada syntax is quite a taxing and trying task. This task get ameliorated by introducing Ada to support programming principles as I do in my courses. One can literary spend a whole semester just teaching the sequential part of Ada. A concerted effort must be made not to get lost in the forest. For example I teach arrays slices to be used to make calls to subprograms with unconstrained array type parameters; this is an example of the implementation and use of abstration. I do not teach all the possible ways to form aggregate expressions. I do not see a principle that can be illustrated with this activity.

Student Population Response.

The student population consists mostly of commuter students who live in the metropolitan area. The great majority of them work while attending classes. From conversations with the students and their comments in the teacher evaluations, one can easily surmise that the difficulty of a given programming course lies in the subject matter and not in the programming language.

The questions and difficulties the students bring to my office are within the same class of questions and difficulties as when I was teaching Pascal. I have yet to meet a student who attributes his or her difficulties directly to the complexity of the language.

The attrition rate using Ada in the core programming courses was not affected. For the first course we had experience up to 50% attrition rate, for the second course up to 33% attrition rate and for the third course no more than a 25% attrition rate when Pascal was the departamental programming language. Many students drop the class due to the fact that they do not have enough time to devote to programming. As I mentioned above, most of the students are full time and hold a job while going to school. We feel that when each student owns or has access to a personal computer the attrition rate will decrease significantly.

Compiler Experience.

During the three years we have been using Ada in our curriculum we have used two compilers. In the fall of 84 we started the teaching of Ada using a pre-validation version 1.3 of the Telesoft Ada compiler. This was not a full Ada compiler.

This fact cause some difficulties. The students could not readily key in program examples from the text book, or try out some of the syntax explained therein. This fact put more work on the instructors as we have to teach the standard features of Ada as well as the version that seemed to work in the compiler we had available. The most notorious features lacking in that compiler version that were noticed in the teaching of the first programming class were the lack of generic io packages of text_io, the lack of some type transfer functions. and the fact that output parameters could be read.

There were many other problems with that compiler most of them due to the incompleteness of the compiler. We felt that the use of the version 1.3 of Telesoft Ada would have lead to very serious problems with the implementation of the second and third programming courses.

In January of 1985, the Telesoft-certified release, Version 2.1 was substituted. This version for VMS was submitted for DoD validation early in 1984 but not validated, failing two programs in the test suite.

With some minor difficulties, this version was used for the spring semester of 1985 in both the first and the second programming courses.

During the spring of 1985, the DEC Ada compiler became available. We have used that compiler from the summer of 1985. This is a full Ada compiler which is supported by the DEC VAX/VMS symbolic debugger. This is an excellent compiler. It generates relatively small object files and the run time of the executable image is more than adecuate for an academic environment.

We currently have the latest validated version 3.10 of the Telesoft Ada compiler. Judging from the list of problems we have encountered with this version (see appendix), it is clear that the compiler to use in a VAX/VMS environment is the DEC Ada; Telesoft Ada is an adecuate choice, but requires large amounts of secondary space to be allocated to each user

Books review.

Below, you find a list of some of the Ada books that have come across my desk or we have used in our courses. I still have yet to find a good book for the introductory class; several books claim to be introductory books; among their faults are poor writing and style; some books do not focus on the teaching of programming from the point of view of the principles of software engineering; some books have many errors regarding either the syntax of Ada or the use of the Ada features. One can find adecuate books nevertheless, but the instructor must carefully read the book to warn students of possible errors or misleading information and the instructor must complement the book with his or her own material. Most books assume at least a minimum of programming experience.

Books aimed to an introductory class.
Introduction to Ada. David Price. Prentice Hall 1984
First lower level book to appear on Ada. Much of the examples give do not give justice to Ada; they are more Pascal like examples. Some of them are actually incorrect. (See page 70 for example). It uses poor examples to show different ways in which an Ada feature can be used. The author breases through the syntax and semantics. Not Recommended.

Introductory Ada. Packages for Programming. Putnam P. Texel Wadsworth Publishing Co. 1986. It introduces Ada and in general programming via packages and subprograms. Since no syntax of statements is given at the beginning of the book, abstraction gets well served. When introducing syntax, it uses cross reference to the LRM. Exceptions and LOOP are introduced early. From there control structures is completed. Types are introduced starting with basic types and finishing with composite types (arrays and records). The book is finished with a closer look to packages. Good and illustrative examples throughout. The book seems adecuate for a beginners class if instructor complements book with his or her own material.

Ada: An Introduction. Sabina Saib. Hollt, Rinehart and Winston. 1985. The book is full of idiotic and erroneous statements (....the safest form of a loop to use is the for

loop..., for example). It coudl be used in an introductory course but instructor must carefully read it ahead to prevent misconceptions.

Books aim to students with programming background.
Most of the Ada books out fall in this category. The large mayority of them teach the sequential part of Ada to a good depth. Most of them gloss over the concurrent features of Ada. The list of books given below can be used for a good indepth introduction to the sequential features of Ada. All books must be complemented with outside material to be used in a class equivalent to CS 2.

Ada for Experienced Programmers. A. Nico Habermann. Dewayne E. Perry. Addiso Wesley. 1983. This book teaches Ada by comparing it with Pascal.

An Introduction to Ada. Second (revised) Edition. S.J. Young. Ellis Horwood Limited, Publisher. 1984.

Programming in Ada. Second Edition. J.G.P. Barnes. Adisson Wesley. 1984.

Introduction to Ada: A Top-down Approach for Programmers. P. Caverly and P. Goldstein. Brooks/Cole Publishing Co. 1986

Ada Language and Methodology. Prentice/Hall International. 1987.

Conclusion.

We feel that the choice of Ada as a departamental programming language has been benefitial to the teaching of programming by having a language that is both modern and supports the needs of modern software development, for th eunification of the curriculum courses with a programming compoenent and ultimately for the student who has the opportinity of exposure to a language that is making a definite impact in our field. We have shown that it is possible to use Ada in the introductory courses with more benefits than disadvantages; and these benefits accrue as the students take the more advanced courses using Ada. It is benefitial to the teaching of programming to use a programming language that is a standard language, with modern features and which is a real language that is making a definite impact in the world.

Bibliography.

[Ada 83] Reference manual for the Ada Programming Language, United States Department of Defense, Washington 1983.

[Augt 79] Austing, R., H. et al., eds. Recommendations for the Undergraduate program in Computer Science, Communications of the ACM, Vol 22, no.3, March 1979, 147-166.

[Auge 83] Augenstein, Moshe, Aaron Tenenbaum, and Gerald Weiss, Selection a Primary Programming Language for a Computer Sciece Curriculum. : PL/I, Pascal and Ada. ACM SIGCSE Bulletin, vol. 15, no. 1, February 1983, 148-153.

[Evans 85] EVans H. et al. Ada as a primary Language in a Large University Environment. Proceedings of the 3rd Annual National Conference on Ada Technology. March 20.21 1985.Pages 7-13.

[SigAda 87] SigAda Education Committee. Academic Liaison Working Group JCollege/University Ada course survey. Ada Letters. Vol VII. No. 2 , March. April 1987 2-99. to 2-101.

Appendix.

The Telesoft Ada compiler has very annoying bugs and characteristics that can render it undesireable or actually unusable

1. Requires large amounts of disc space. 5000 blocks is not unusual.

2. Can not instantatiate generic packages.

3. Files produced by the compiler are huge. For a "one line program", the DEC Ada compiler produces a five block file. For the same program the TeleGen2 compiler produces a two hundred and eleven block file!

4. Text input is hopeless.

5. There is no debugger.

6. The DELETE operations on the Ada library only removes the names form the library. It does not make the library any smaller.

7. The FORMAT utility is useless.

This page left blank intentionally

This page left blank intentionally

This page left blank intentionally

6.'

Introducing Ada® and Its Environments into a Graduate Curriculum

by

Major Patricia K. Lawlis and Karyl A. Adams
Air Force Institute of Technology
Wright Patterson AFB, Ohio 45433

Abstract

At the First symposium sponsored by the Ada Software Engineering Education and Training (ASEET) team, there was a presentation discussing the growing commitment at the Air Force Institute of Technology (AFIT) to the Ada language and philosophy [Lawlis, 1986]. This paper is, in a sense, a continuation of that presentation. Since the time of the last symposium, AFIT's curriculum has continued to reflect continuing use of Ada. Improved facilities, expanded course work, and greater emphasis on research have each contributed to the expanding role of Ada in educating the AFIT student.

Understanding the growing need within the Department of Defense (DoD), AFIT has accepted the challenge to educate Ada professionals. To that end, AFIT strives to provide its officers with the most advanced information available regarding the language and its uses, particularly as it applies to military systems and applications. The Institute's computer faculty has taken an active role in defining and constantly improving the Ada curriculum in order to reflect the current state of the Ada technology and research.

In concert with the DoD's philosophy that Ada is more than the language itself, the curriculum does not isolate the language. The approach taken is to present the language within a framework of the software engineering principles it embodies, the software development environments that should support Ada development, and the real time arena in which Ada must solve problems. This comprehensive approach will better prepare the AFIT student to assume a productive role within the Ada community.

New courses have been developed for specific areas of software environments and real time design and development issues. Existing

courses have evolved to present a more complete view of the language within a practical context.

From all indications, the inclusion of the the advanced course work, coupled with increased use of the Ada language throughout the curriculum, has been a critically important step in the maturation of the AFIT graduate computer program. Students and instructors alike have acknowledged the value of the approach. Accompanying the maturing course materials has been an increase in Ada-based research. As the program has matured, the level of research has reflected that maturity with current thesis efforts investigating facets varying from compiler performance measures to the study of internal interfaces within proposed Ada support environments.

It has not been trivial to convert an entire graduate program to a new programming language. There have been failures or setbacks, to counter successes. However, the results of the Ada effort at AFIT have been encouraging and generally positive. Enough so that it is important to present them, and to share those successes and failures that accompany such an undertaking.

This paper provides a look at the current AFIT program. It discusses the roots of the program and describes how the AFIT program has emerged over the years. In conclusion, the paper states the impact of incorporating Ada as the foundation language for the curriculum.

Introduction

The introduction of Ada into an established computer science and engineering curriculum is a difficult task. It requires insight, dedication, and time to realize success. Faculty members at AFIT recognized the need for Ada education. Moreover, they realized AFIT's potential as a DoD educational resource and began to incorporate Ada into the graduate curriculum around 1980.

This first venture with Ada was a small one and met with mixed success. Lack of appropriate support facilities, most notably a compiler, made the early efforts rather adventuresome. Encouraged by the DoD's continuing support for the language, AFIT continued its efforts and gradual growth in the facilities led to improved use for the language.

AFIT today supports a relatively robust Ada based curriculum. In addition to fundamental software engineering courses taught with Ada

many of the advanced courses in traditional computer science areas, such as graphics, compiler theory, operating systems, and database theory, either use Ada exclusively or for significant project work. Several new courses have been brought into the curriculum in response to the technical areas so vital to Ada's complete use, such as software support environments and real-time systems. A look at the several year process will serve to define how AFIT developed its Ada capabilities.

The Early Years

The early experiences with using Ada at AFIT were undoubtedly much like those of other universities. With the language being so new, there were few who knew enough about it to instruct. There were few textbooks, no compilers, and very little support in the commercial world when AFIT decided to venture into the Ada world. The language definition hadn't even stabilized yet. It was a venture that only the strong-hearted would undertake.

However, during that time, the interested faculty at AFIT developed a first course in Ada. With no access to Ada compilers within the institute, the language could be studied only from a theoretical standpoint. Exercises in programming Ada code were "hand compiled" by the instructor. This was certainly an adventure for student and instructor alike! An Ada course was taught several times in this fashion before any support facilities were available. This lack of computer support certainly detracted from the overall effectiveness, and contributed to a general feeling of "incompleteness" in the course. Despite the difficulties of instructing this powerful new language in a virtual support vacuum, Ada had been introduced at AFIT. The potential of the language was recognized and the impact of Ada began to spread to other courses.

The first courses outside of the actual introductory Ada course to reflect the influence of the language were those in the compiler sequence. Long before AFIT possessed an Ada compiler, students were challenged to delve into the language definition as part of the compiler courses. Project work involved the development of partial compilers for the language in Pascal. By contrasting the requirements which Ada, versus less powerful languages such as Pascal, levied on compiler writers, greater appreciation of the technical composition of the language was achieved.

Gradually, other events began to unfold at AFIT which enhanced the fledgling Ada program. A most significant facet of the infant program was

the beginning of Ada-based research for the master's thesis work. None of these early theses could make use of Ada as the language for development, but they did look at several of the technical aspects of Ada and its necessary support environment. Students investigated such areas as compiler design, debuggers, editors, and mathematical support libraries which would exploit and support the features of the language [Garlington, 1981; Gaudino, 1981; Ferguson, 1982; Lawlis, 1982].

Thus the first days of Ada at AFIT were a struggle. But the potential for Ada was recognized, and interested faculty members continued to push for its continued use. Those who understood the potential for Ada continued to expand the role of Ada at AFIT. They pushed for better computing support and for an expanded curriculum for the computer science and computer engineering students at the institute. Faculty members also became active in the Ada community and established AFIT's membership in task teams established by the Ada Joint Program Office (AJPO). Slowly, but surely, their efforts began to pay dividends.

The Impact of Compilers

With the advent of available Ada compilers, the AFIT program was ready to move forward with the rest of the Ada community. The first compiler acquired at AFIT was an early, unvalidated version of the TeleSoft Ada compiler, along with its TeleQuiz package, which ran on a VAX under the Berkley 4.2 Unix™ operating system. Although the compiler had not yet passed validation testing, it did provide AFIT with a much needed support facility.

With the receipt of a compiler, the manner in which the Ada-based courses were taught changed. The introductory Ada course could embrace the practical aspects of using the language in addition to the theoretical studies. Compiler projects could now be accomplished in Ada instead of other programming languages.

While the TeleSoft system provided a heretofore non-existent capability, the most significant impact on AFIT's program was the acquisition of two noteworthy, validated compiler systems. In early 1985, both the Verdix Ada Development System (VADS) for Unix and the DEC Ada system for the VMS™ operating system were brought into the institute. These two compiler systems, coupled with the TeleSoft system provided AFIT with the basis for complete instruction in the language

features, drill exercises with the TeleQuizzes, a basis for compiler comparison, and complimentary facilities under both Unix and VMS.

A Changing Curriculum

Going from a dearth of capability to such relative prosperity opened many new avenues for Ada education at AFIT. During this timeframe, the interested faculty at AFIT developed and proposed a radical change to the entire computer science/engineering curriculum. This change highly recommended that Ada be used as the "official" language for the computer science and engineering students, completely replacing Pascal and C in that educational series.

Such a proposal had severe implications for the existing curriculum, as it would in any organization. Two departments within the institute cooperatively administer and instruct the graduate computer science (GCS) and graduate computer engineering (GCE) curricula. Agreement within one department for such a severe departure from the status quo can be difficult. The difficulty with devising a reasonable plan which could satisfy both departments' needs was significant.

In the final telling, the curriculum committee accepted the proposal to incorporate Ada as the basis for the GCS/GCE program. The class of students entering AFIT in June of 1985 was the first class to start the Ada-based program. In order to smooth the transition period to Ada and to satisfy departmental requirements that well established courses not be adversely impacted by mandating an immediate conversion to Ada, only the introductory programming courses at AFIT were completely redefined. That really was all that could be handled by the Ada literate faculty anyway. Only three faculty members had any significant background with Ada and they formed the core instructional pool for the Ada-based courses.

The introductory course was again revamped to focus more on the software engineering aspects of the language. Programming exercises were expanded to make full use of the available compiler support. Armed with a conviction that Ada could, and should, make a difference at AFIT, the faculty prepared to greet the incoming class of students.

Growth of the Ada Program

In the first class of students to start the Ada sequence there were sixty students. For a faculty still relatively inexpert with Ada, and certainly not fully comfortable with the new compilers, this was a large class to manage. The students were divided into three sections and the first major Ada instructional effort was underway.

One problem area that immediately surfaced was the computational load on the computer equipment when numerous Ada students were developing software simultaneously. The system was known to handle up to forty students , but the detrimental impact on system response time was felt by both the Ada students and all other AFIT users. This first class quickly demonstrated the seemingly enormous space and time requirements that Ada development systems require.

Satisfactory resolution of the facility scheduling and resource allocation problem, to fully support Ada development while not hampering other equally important programs, became an on-going effort within AFIT. While no complete solution was found, by attempting to balance the computational load on the centrally available systems, increasing the available computational power, and trying to encourage good, common sense usage, the situation was at least tolerable.

Students and instructors alike survived this first session of Ada education and it was difficult to determine who learned more. Prevailing sentiment was that the instructors emerged much the wiser for the effort. In addition to honing their instructional skills, faculty members had successfully interested a significant group of the students to undertake Ada related research work.

The research that was undertaken by this class of students provided the much needed impetus for more AFIT faculty members to become active participants in the Ada curriculum. Most of the remaining twenty instructors had maintained a somewhat ambivalent posture toward the use of Ada. Within the advanced courses there had been no compelling reason to introduce the new language, so it was not used. Ada had been seen by many as just a language being taught within one department, but not to be used within the other.

These new "Ada" students advancing through the curriculum, and their research, forced the next major advancement in the acceptance of Ada into the curriculum. Faculty found it more necessary to be conversant

with Ada to serve as advisors and committee members for thesis research. Some were "stunned" to discover that students did course projects in Ada because it was the only language they knew! In response to this heightening faculty need for Ada education, a course for the faculty was conducted. The course was well attended and served to improve faculty awareness and understanding of the capabilities and features of the language. As a result of this experience, more of the AFIT faculty were beginning to see the potential for Ada and the need for a more consolidated curriculum. There was much work to be accomplished before Ada was fully incorporated into the GCS/GCE curriculum, but the foundation had been established.

The response from the first class was positive. Although Ada was quite different from other languages with which the students were familiar, they recognized the importance of understanding the DoD sponsored language. Many enjoyed learning and using Ada and felt it would be very useful in future Air Force careers. Once the first class completed the course, AFIT instructors were busy improving the content based on student critiques. By the time the next class of students entered AFIT in June of 1986, the Ada courses included a much revised introductory sequence, better integration into advanced computer science courses, and advanced courses in software environments and real time operations.

The actual instructing of the language was still accomplished entirely within one department. The vast majority of the courses which used Ada for project work were also offered out of the one department. Student critiques of the program, their desire to continue to use the language they were taught instead of learning a second or third and new faculty members who had a working knowledge of the language were causing a slow infusion of Ada throughout the curriculum. Instructor involvement improved and more faculty were attempting to use Ada as an integral part of their courses. Another course for faculty members was offered, and several faculty attended the formal classes with the students. While there were still many roadblocks to a completely integrated Ada curriculum, slow acceptance was at least beginning.

AFIT's Current Curriculum

The current curriculum at AFIT attempts to provide essential Ada skills to computer scientist, engineer, and manager alike. Ada literacy is the minimum goal for all of the GCS/GCE students. These students are

taught Ada as their first formal language. Other engineering students in electrical engineering specialties also learn Ada during their first few months at AFIT.

The introductory program has been defined to accommodate the computer specialists as well as other engineering students. Two Ada courses are offered, each tailored to the specific needs of the identified educational path. One course is the "Introduction to Software Engineering with Ada". This course is mandatory for all GCS/GCE students. This is a comprehensive course designed for the experienced student. It covers the full extent of the language features. The focus is not on Ada syntax, but rather on the embodiment of software engineering principles within the language. Students are expected to be proficient in the use of the language, understand the philosophy and rationale behind the language and its support environments, and be able to reasonably design and develop small Ada code samples. The course also serves as an introduction to basic data structures, with many of the programming examples illustrating the language features which support the definition of such structures. The course has been designed as a one quarter course with four hours of lecture and a three hour lab session each week [ENC, 1986].

The second introductory course is much less rigorous than that for the GCS/GCE students. This course has been designed for those engineering students who take some of the same advanced courses with the GCS or GCE students and need to have a fundamental knowledge of an implementation language. These students do not need the in-depth knowledge that the computer track students require.

The alternative course is an "Introduction to Computer Science". The course is designed with the inexperienced student in mind. It does include very basic software engineering tenets and rudimentary data structures. It makes no attempt to cover the entire Ada language however. It is a much gentler introduction to the language, as is reflected in the choice of textbooks, the introductory text by Texel [Texel, 1986]. While the course is not as rigorous as its companion course, the students are expected to become knowledgable and proficient in the use of basic Ada constructs. Ada will become their language of implementation, and they must be able to satisfactorily construct working Ada programs [ENC, 1986].

With the introductory course completed, the frequency with which a given student will see or use Ada during the remainder of the AFIT stay is variable. Although many factors strongly push for a more consistent use of Ada in the program, parochial interests have continued to support a

house divided. There are advanced sequences and courses available to the GCS/GCE students which make exclusive use of Ada in support of the course materials.

The two sequences which are completely Ada based are in the areas of compiler theory and computer graphics. As discussed earlier, the compiler sequence has used Ada as its focal point for several years. Contemporary offerings have used Ada as the language of implementation for partial Ada parsers and compilers. The graphics sequence is a recent addition to the list of courses using Ada as its chosen language for implementation. It has taken some time to prepare an adequate basis of device drivers that permit reasonable use of Ada as the programming language.

Courses which exclusively use Ada are those courses which have been specifically devised to study Ada related topics. To date, there are two such courses which have been fully defined. The first, a course in "Advanced Software Environments", has been immensely successful. This course examines the STONEMAN [DoD, 1980] model as an example of a programming environment definition. Students then analyze other existing environments. Although Ada environments provide the nucleus for such as study, other environments which have had significant impact in this area are also examined.

This course has been offered twice in the past year and each time has been well attended. Students have enjoyed the course and several have selected thesis topics based on material covered in the course. Class software projects have resulted in a rudimentary prototype Ada Programming Support Environment (APSE) which has served to improve AFIT's Ada support. This environment has also provided a basis for future research and analysis for class efforts and individual thesis work.

This prototype APSE, called the AFIT Research Concept for an Ada Development Environment (ARCADE), takes the approach of providing an easy and uniform interface for accessing AFIT's existing Ada support facilities (VADS and the DEC Ada systems). Its use does not require knowledge of the underlying operating system or the Ada support system Thus it has been a useful tool for the latest group of Ada programming students [Austin, 1986a; Braaten, 1986; Linski, 1986a].

The second advanced Ada course is one in "Real-Time Software Systems". This course is scheduled for its debut in 1987. The intent of the course is to focus on the issues unique to real-time processing and to

analyze how Ada addresses them. Hopefully this course will meet with the same success as the environments course has. Defining and preparing the course has been a challenge, so the results from its first offering are eagerly anticipated.

There are additional courses throughout the curriculum that use Ada to some extent. The operating systems, data structures, and database courses are beginning to mandate its use for at least some project work. In some classes language choice is arbitrary and left to the student's preference. Other languages are mandated in some advanced courses, sometimes for rather arbitrary reasons. It is this area of inconsistency within the program that still remains to be satisfactorily addressed.

A discussion of the AFIT curriculum would be incomplete without some reference to the research that compliments it. Thesis research is an integral part of the AFIT masters program. All candidates must successfully complete an approved thesis effort in order to receive a degree. Such work must demonstrate the candidate's integration of knowledge from several course areas. The Ada research at AFIT has grown through the years as the courses themselves have. This research serves to compliment, and often enhance, the classroom environment.

The Value of Ada Research

While it has been rewarding to see the curriculum at AFIT expand coverage of Ada and related technologies, the most exciting area of growth has been the Ada research. Since 1983 AFIT has supported Ada related research. The nature and number of the research projects has increased dramatically since that time. As Ada support facilities have become available, research has broadened to include both theoretical studies and practical applications. Students are investigating such areas as compiler research and performance evaluation. AFIT's software and application areas such as graphics [Wo__, __, Orndorf 1983, Ruegg 1983, Hanson ____]. As research has touched other areas of AFIT's curriculum, ____ have been started [Austin 1986].

As the research has expanded, students ____ outside support for their efforts from the AFSF ____ E&V team and the Air Force Wright Aeronautical Laboratories. This support has been the source of ____ specialized equipment and better ____

These organizations, as well as AFIT instructors, have helped to define the rudiments of an Ada research program that should produce a focused series of research projects for the next several years.

As Ada resources have matured and become available at AFIT, the potential for research has improved. Some of the very recent research has dealt with the most contemporary of Ada issues. Several of the topics from these efforts are related to the issues associated with software support environments. Such efforts have done much to strengthen AFIT's research program and to provide a focus for future work.

One such thesis looked at the issues involved with the interface definitions within an APSE. Working with an available partial implementation of the Common Apse Interface Set (CAIS) (DoD, 1985), the student was able to integrate the CAIS with ARCADE, the prototype APSE that had resulted from the environments class. The student also transported the original APSE from one operating system (Unix) to another (VMS) while maintaining identical function within each version. This work provided needed empirical data regarding how well APSEs can be transported and how the interface issues impact the selection, design, and integration of tools within the environment (Linski, 1986).

Other recent theses investigated the definition of Ada interfaces to provide improved access to existing graphics libraries and to provide design tools to assist in the creation of human computer interfaces. Both areas are of increasing interest in the Ada world as workstations with improved graphics capabilities become more commonplace. One thesis effort designed and implemented an extensive Ada pragma interface into an existing commercial library which defines the Graphical Kernel System (GKS) in FORTRAN subroutines (Hanson, 1986). The effort maintained compliance with the draft standard for the Ada language binding to the GKS standard. This permits a well defined standard interface from Ada into a powerful support facility without requiring that the library be rewritten in Ada. Such a technique could be a very satisfactory short term solution to designing large Ada systems without incurring the overhead of rewriting existing, proven software libraries. The other effort investigated user interface issues within the APSE definition, an area that is particularly vague in STONEMAN. The results of this thesis were a set of design tools which could be used to prototype user interfaces in a consistent manner (Cochran, 1986).

There have certainly been numerous thesis efforts of note at AFIT, far too many to discuss at any length here. Topic areas receiving

continued attention in future projects include the environments area, compiler design and performance issues, graphical applications, hopefully educational issues, and concurrent and distributed processing with Ada. A healthy research program should be indicative of the program's overall health, as the research could not progress without strong course support.

Ada's Future at AFIT

As it is with any growing thing, the future of AFIT's Ada program is difficult to predict. The last several years has seen the curriculum grow from its humble beginnings into a stable program supporting the education of Ada professionals. Both the fundamentals of the language and the concepts of the intended use and support of the language are emphasized. Practical use of the language to support traditional computer science areas is progressing. The progress seems frustratingly slow, but it is progress that can be measured over time.

The maturation of the program has taken a significant step forward with the addition of the advanced courses and the improved research. With the introduction of the environments course, and it is hoped the real-time course, a dramatic cyclic effect has been noted in the interaction of classroom work and research. The research topics have actually provided facilities that can be used to support classroom projects and analysis. As these research facilities are used by classes, improvements are suggested which have led to additional class projects and thesis topics. This can be a very healthy cycle if it is managed appropriately.

AFIT's involvement with AJPO activities, most notably the E&V and ASEET teams, has made AFIT's program and research more visible. This is important for the continued growth of the program. Faculty members on such teams are able to stay current with important Ada activities. This feeds back to classroom material and provides the student with up to date information in this rapidly growing area.

Overall the growth has been good, but slow. The critical elements in slowing the introduction of Ada into the curriculum have been misconceptions over the language, strong adherence to parochial interests, and the loosely defined interrelationship between the two departments responsible for the education of the GCS and GCE students. Until such time as the departments can consolidate their opinions, the students will continue to see a program that is not as strong as it could be.

The departments are moving in the proper direction however. A new working group, whose principle aim is to define a consolidated research program, is actively working to open the channels of communication for the betterment of the entire program. These involved faculty members are actively increasing the use of Ada in their courses and encouraging others to do so also.

The future of Ada at AFIT should be bright. The school has made a commitment to adopt Ada as the language for the computer curricula. Being a DoD resource, this is not a commitment that can be taken lightly. The increasing research, along with the dedication of the faculty members, will keep Ada a viable part of the curriculum well into the next decade, and perhaps beyond.

BIBLIOGRAPHY

Austin, Capt Kathy et.al., "ARCADE - System Requirements",
Unpublished report, Air Force Institute of Technology, 1986.

Austin, Capt Kathleen M., *Applying Ada Programming Support
Environment (APSE) Concepts for Computer Integrated
Manufacturing Systems (CIMS) Solutions*, Masters Thesis,
AFIT/GLM/ENC/86S-2, Air Force Institute of Technology,
September 1986.

Braaten, Capt Alan J. et. al., "ARCADE - Design and Implementation",
Unpublished paper, Air Force Institute of Technology, 1986.

Cochran, Capt Mark A., *A User Interface Toolset for Ada
Programming Support Environments*, Masters Thesis,
AFIT/GCS/MA/86D-2, Air Force Institute of Technology,
December 1986.

Craine, Capt David., *Toward an Ada Compiler Evaluation Capability*,
Masters Thesis, AFIT/GCS/MA/86D-3, Air Force Institute of
Technology, December 1986.

Department of Defense., " Proposed Military Standard Common APSE
Interface Set (CAIS)", 31 January 1985.

Department of Defense., "Requirements for Ada Programming
Support Environments (STONEMAN)", February 1980.

ENC, Instructor's Handbooks, Course descriptions for classes taught in
the Department of Mathematics and Computer Science, Air Force
Institute of Technology, 1986.

Ferguson, Capt Scott E., *A Syntax Directed Programming Environment
for the Ada Programming Language*, Masters Thesis,

AFIT/GCS/MA/82D-1, Air Force Institute of Technology,
December 1982.

Garlington, Capt Alan R., *Preliminary Design and Implementation of
an Ada Pseudo-Machine*, Masters Thesis, AFIT/GCS/MA/81M-1
Air Force Institute of Technology, March 1981

Gaudino, Capt Richard L., *Analysis and Design of Interactive
Debugging for the Ada Programming Support Environment*,
Masters Thesis, AFIT/GCS/MA/81D-3, Air Force Institute of
Technology, December 1981

Hanson, Capt Markoe S., *An Application of Advanced Ada Language
Features to Data Structures in a Graphics Programming
Environment*, Masters Thesis, AFIT/GCS/MA/86D-4, Air Force
Institute of Technology, December 1986

Lawlis, Capt Patricia K., *Arbitrary Precision in a Preliminary Math
Unit for Ada*, Masters Thesis, AFIT/GCS/MA/82M-2, Air Force
Institute of Technology, March 1982

Lawlis, Major Patricia K., Slides for ASEET Symposium Presentation
June 1986

Linski, Lt David M., "ARCADE An In Depth Analysis", proposed
technical report prepared as a special study, Air Force Institute
of Technology, December 1986

Linski, Lt David M., *Investigation of the Common APSE Interface Set
(CAIS)*, Master Thesis, AFIT/GCS/MA/86D-5, Air Force Institute
of Technology, December 1986

Orndorf, , *Evaluation of Automated Configuration Management Tools
in Ada Programming Support Environments*, Masters Thesis,
AFIT/GCS/ENC/84, Air Force Institute of Technology, 1984

Ruegg, Lt Raymond S., *AFIT GKS A GKS Implementation in the Ada Programming Language*, Masters Thesis, AFIT/GCS/MATH/84D 5, Air Force Institute of Technology, December 1984

Texel, Putnam P, *Introductory Ada Packages for Programming*, Belmont, California Wadsworth Publishing Company, 1986

Witt, Capt Donald J, *Using Ada in the Real Time Avionics Environment Issues and Conclusions*, Masters Thesis, AFIT/GCS/MA/85D 6 Air Force Institute of Technology December 1985

Applying Formal Specification Techniques in
an Ada-based Software Engineering Environment.

Charlene M. Hamivka and Laurence J. Latour
Department of Computer Science
University of Maine at Orono
Orono, Maine 04469

Abstract

Of concern in this paper is the application of existing "black-box"
module specification techniques in a practical software engineering
environment, and the integration of these techniques into a university
software engineering curriculum utilizing Ada as the primary design and
implementation language. A three step process is proposed as an aid to
applying these techniques in the design and implementation of a
non-trivial software system. It consists of (1) formal "black-box"
specifications of each internal software module, (2) a specification of
external system behavior, and (3) precise mappings between the two. The
design of a simple line editor is presented as an example.

Introduction

Formal specifications of a software module have been the focus of much
attention from the software engineering community. If a module's
specification is precisely defined both syntactically and semantically,
there should be no errors that occur as a result of either an implementer
or user misinterpretation of the specification. We have been presenting
existing work in formal specifications as part of our software
engineering course, but have encountered some difficulty in applying
these theories to systems developed in class. We observed that modules
interact with each other in one of two ways, regarding the flow of
information between them. We refer to these as 1-way or 2-way
interactions and show that the type of interaction influences the formal
specification approach.

This paper presents both the reasons for teaching formal specifications
as a part of a software engineering course and some of the difficulties
we have encountered by trying to apply the present principles to complex
systems. It is organized in the following manner. The first section
contains a review of formal specification work done by Parnas, Guttag,
and others. The second section describes our class environment and
objectives. In the third section, we present our experience in applying
formal specification techniques to existing systems and our observations
of module interaction. We propose a method of formally specifying a
software system that accounts for these interactions. The need for
teaching formal specification theory is discussed in the final section.

I. Review of Formal Specification Theory

According to Parnas [1], the goals of a specification scheme are as
follows:

1. The specification must provide to the user all the
    information that is needed to use the module correctly
    and nothing more.

2. The specification must provide to the implementer all
    the information that is needed to complete the module
    and nothing more.

3. The specification must be sufficiently formal that it
    can conceivably be machine tested.

4. The specification should discuss the module in terms
    normally used by the user and implementer alike.

The need for formal specification in a software system may not be
immediately obvious. The work done in this area by Parnas [1,2,7],
Guttag [9], and others however, show that semantics of an operation need
to be specified just as precisely as the syntax, otherwise the user may
obtain an unexpected result.

The following is an example of an Ada package specification :

```
package STACK_PACKAGE is

    type STACK is private;

    procedure PUSH (X : in INTEGER; S : in out STACK);
    procedure POP (S : in out STACK);
    procedure TOP (X : out INTEGER; S : in out STACK);

end STACK_PACKAGE;
```

Figure 1. Stack package specification

This appears to be the specification of an unbounded stack, but suppose
it is the syntactic specification of a stack that holds a maximum of 124
items and overflows by dropping the lowest item (item#1) off the bottom?
(Bartussek and Parnas [2].) Syntactically, the Ada specifications of
these modules are identical, yet their semantics are very different.
English comments have been the usual method of conveying the semantics,
but they are subject to misinterpretation. In using formal
specifications, we attempt to provide a precise and unambiguous
description of the module's behavior to both the module implementer and
user.

We have chosen here to define the semantics of the overflow stack using
Parnas' assertions on traces. A trace is simply a recording of the
operations performed on a module, in this case a sequence of push, pop,
and top operations.

Assertions are typically partitioned into three categories: (1) those
that define a base set of normal form traces, (2) those that define a set
of trace equivalences, allowing us to convert an arbitrary trace to some
normal form trace, and (3) those that define the value of functions
applied after some normal form trace. Together, these assertions provide
a precise definition of the module's behavior. (For a similar "black
box" approach to module specifications, see Guttag [6,8].) Figure 2
illustrates this method. The corresponding specification written in
English would be much more difficult.

```
package STACK_PACKAGE is

   type STACK is private;

   procedure PUSH (X : in INTEGER; S : in out STACK);
   procedure POP (S : in out STACK);
   procedure TOP (X : out INTEGER; S : in out STACK);

-- Legality:
--     For all T, $\mathcal{L}$ (T)

-- Equivalences:
--     0 < N < 124  ==>
--         PUSH$^N$(a$_i$).POP = PUSH$^N$(a$_i$)
--         PUSH(a$_i$).PUSH$^N$(a$_j$) = PUSH$^N$(a$_i$)
--         T.TOP = T
--     N > 0  ==>
--         POP$^N$.PUSH(a) = PUSH(a)

-- Values:
--     V(T.PUSH(a).TOP)  = a mod 255

end STACK_PACKAGE;
```

Figure 2. Stack package with formal specifications

Formal specification theory is still in the development stage in that to
date, most of the work has dealt with small systems. In attempting to
apply the theory to the more complex systems implemented in our class, we
encountered instances where a module's behavior was not so easily
specified. We will discuss one such example, a line editor, in the
following sections.

II. Class Environment and Objectives

We have developed a series of software engineering courses as part of our
undergraduate/graduate program at the University of Maine at Orono. In
them, we present software engineering principles using Ada as both a
design and implementation language, since its package structure is well
suited towards a top-down, information hiding approach to system design.

Since a great deal of attention has been focused on the issues of formal specification, the course material includes extensive coverage of this topic. While an Ada specification can be used to abstract away implementation details from the user, it also abstracts away the semantics of an operation as well. We present formal specification as a method of supplementing the Ada package specification. In addition to the syntax, we attempt to specify the precise meaning of the package interface.

We assigned a line editor to the class, to be implemented in Ada, as their software engineering project. After choosing an appropriate system design for the editor, module specifications were handed out and implementations of the modules were written for the specifications as they existed. No modifications were allowed. The editor consists of the following main packages:

1. COMMAND_MODULE - performs a lexical scan of the input stream and parses it into an abstract data type (ADT), COMMAND, with a varying number of parameters.

2. DRIVER - calls COMMAND_MODULE to get a command then passes this command to the OPERATIONS package.

3. OPERATIONS - After receiving a command, it calls COMMAND_MODULE to get the various parameters associated with the command. When these are obtained, it performs the desired operation on a document.

4. DOCUMENT_PACKAGE - contains the file to be edited.

5. TERMINAL - low-level output module.

The layout of the system is illustrated in figure 3. In this and all subsequent diagrams, we are using the system representation of Buhr [5]. Packages (modules) are represented as rectangles, calls to other packages are shown by arrows connecting packages, and data flow is the smaller symbol, O➔

III. Methods and Observations

Previous work in this area has been applied towards small modules such as the bounded stack package (Figure 2). However, when developing formal specifications for our software engineering class, we quickly encountered instances where a more comprehensive view was needed.

By studying the modules and the relations between them, we observed that a module and its users have either a 1-way or 2-way interaction. A 2-way interaction has the users supplying input to a procedure and receiving some output back. This allows verification that the results received are in accordance with the information sent in. The stack package referenced earlier is an example of this type of interaction.

82

```
                        --------
         ---------------| DRIVER |----------------
        |               --------                 |
        |                                        |
        |                                       ! ♀
        |                                        |
       δ|                                        v
        |                                    ------------
        |                                   !            !
        v                                  ! OPERATIONS !
    ----------                             !            !
   !          !                            ------------
   ! COMMAND_ !<----O*--------------------- !    !
   ! MODULE   !           -------------------    ! ♀!δ
   !          !                             ♀    !
    ----------                                   !
        |                                        v
       ! ♀                                  ----------
        |                                  ! DOCUMENT !
        v                                   ----------
    ----------
   ! TERMINAL !
   !          !
    ----------
```
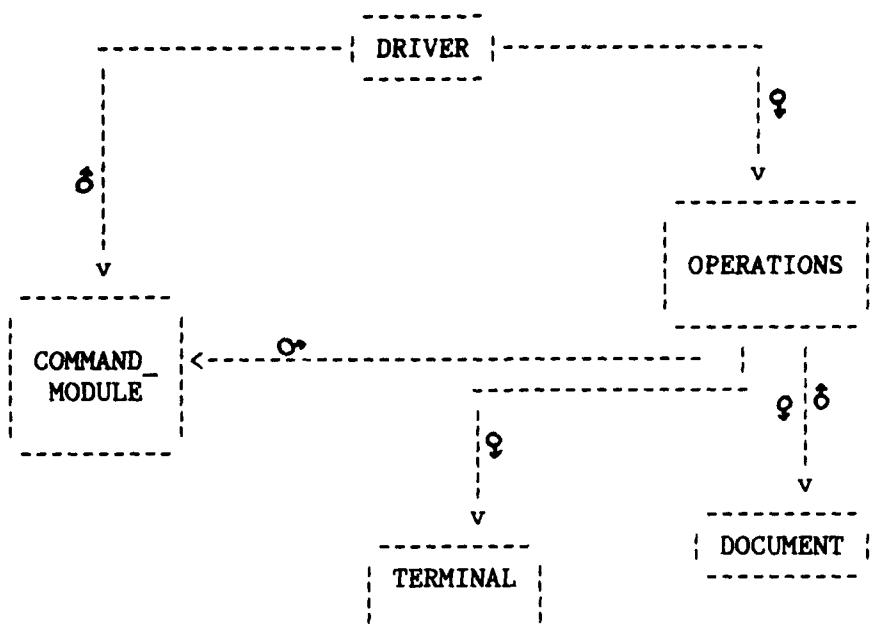
Figure 3. Line editor package layout

In a 1-way interaction, the users either receive or send information to
the module, but not both. A 1-way interaction occurs in the case of an
I/O package, such as COMMAND_MODULE. The package accesses a low-level
input stream and transforms it into an abstract command object, which is
then sent to the user who requests it. Since the user has no access to
the input stream, there is no way to verify that the command received is
the correct result of the input. Furthermore, attributes of this command
may be passed to an entirely different package which then performs an
output operation to a low-level device. Again, there is no way to verify
the output matches that of the input. An example of these interactions
is shown in figure 4.

Our line editor has modules with both types of interactions. Modules
involved in the 2-way interactions could be specified by the techniques
previously discussed; however, these techniques were not adequate when it
came to formally specifying modules with 1-way interactions.

Instead of viewing our editor as a standard, top-down, layered system
depicted in figure 3, we used the idea of a bubble. (See figure 5.) A
bubble is drawn around the modules and the notions of top and bottom
levels are removed. The end-user interface is represented by the outside
edge of the bubble.

The details of the system are encapsulated by the bubble and unseen by the end-user (which is a person in our case, but could also be a machine or another software module). Packages which touch the edges of the bubble are gateways into and/or out of the system and usually represent some type of I/O package.

We chose this system representation because it more closely depicts the actual environment. The end-user receives output from the system which can only be verified based upon previous user input. Based on this system representation, we developed a specification technique which can be broken down into three steps.

Step 1. Specify the end-user interface. This is the only place in our editor where the total system behavior can be verified, meaning that the output received is the correct result of the input sent in. The end-user interface specification should provide an overall view of the system and state precisely what the function of each operation is. Any packages that interact with the "I/O" packages must use this specification in order to fully understand how to utilize the information it receives.

Step 2. Specify the internal modules of the system (all the packages inside the bubble). For all modules with a 2-way interaction, specify their behavior according to the techniques proposed by Parnas or others. For modules with a 1-way interaction, we must assume, at this point, that all information given to the users of these modules is correct. For example, in our COMMAND_MODULE, we assume the lexical scanner and parser work correctly in transforming the input stream to a command. What we do specify is the correct syntactic form of a command. This allows us to state what parameters are legally called for any command. Figure 6 shows a partial specification of our COMMAND_MODULE. If all modules have been correctly specified, any errors occurring at this point should be due to one of the "gateway" packages (1-way interactions) operating incorrectly.

Step 3. A mapping is provided between the end-user interface and the gateway module specifications. Once this is done, we no longer need to assume these modules work correctly.

Conclusion

People normally have some intuition about a module's function, and formal specifications are our means of making this intuition precise. By stating the syntax and the semantics of a module precisely, implementers and users of a package are prevented from having different views of the module's behavior. While the traditional specification techniques work fine for packages with 2-way interactions, we found they broke down when trying to apply them to packages with 1-way interactions. To deal with this, we developed a 3-step approach to formal specification of a system which seems to work reasonably well for our line edit r. We realize this example, though non-trivial to a student, is still small compared to existing production software. For this reason, we plan to experiment further with this methodology on a collection of environment tools

initially proposed in [9] and developed by a senior/master's project group here at the University of Maine.

Overall, our experience of presenting and applying formal specifications went well. The need for formal specification was generally well accepted by the students, especially after they encountered errors due to misinterpretation of English language specifications of package semantics. We realize current specification techniques are still in the early stages of development, but they are useful and will become an important aspect of system design.

Acknowledgements

2-way interaction   1-way interaction
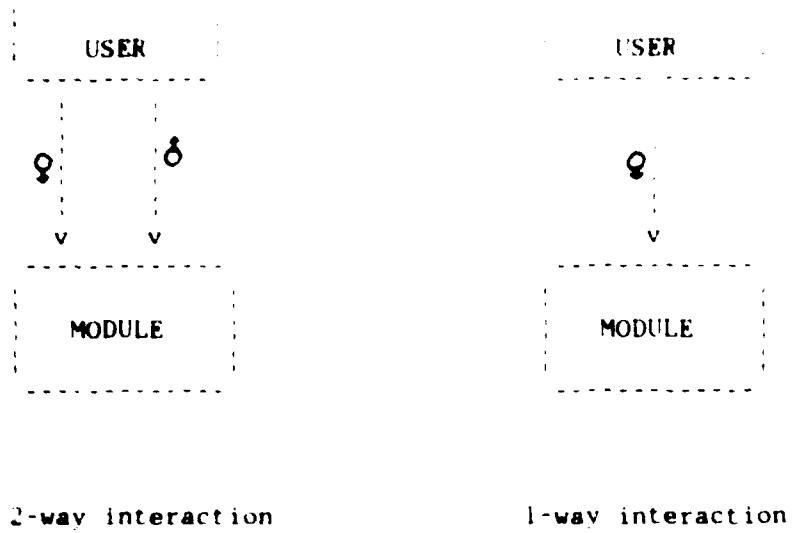
figure 4. Package interactions
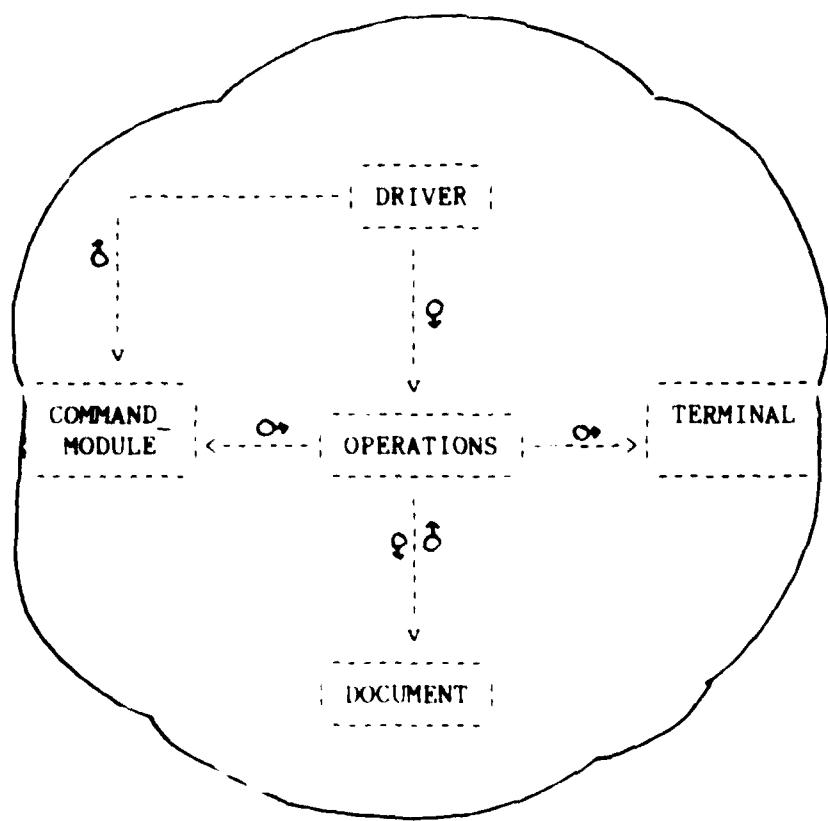
Figure 5. Bubble layout of line editor

```
package COMMAND_MODULE is

    type COMMAND_TYPE is (APPEND_CMD, CHANGE_CMD, DELETE_CMD,
                         (...., SUBSTITUTE_CMD);

    function COMMAND return COMMAND_TYPE;

    function LINE1 return LINE_NUMBER;
    function LINE2 return LINE_NUMBER;
    function LINE3 return LINE_NUMBER;

    function ORIGINAL_PATTERN return STRING;

    function NEW_PATTERN return STRING;
        .
        .
        .


--    Legality:  any command is legal and any series of commands is legal

--       L(COMMAND)
--       L(T) ==> L(T.COMMAND)

--    Equivalence:

--    A command followed by any parameters is equivalent to a command
--    by itself (calling a parameter does not change the command type).

--       L(T.F) ==> T = T.F

--    A command parameter is legal if a valid command type is input and
--    the function belongs to the set of parameters for that command
--    (stated by the FUNCTIONS_OF call

--       L(T.COMMAND.F) ==> (V(T.COMMAND) = COMMAND_TYPE &
--                           F ∈ FUNCTIONS_OF(COMMAND_TYPE)

--    The legal parameters for each command are stated below:

--       FUNCTIONS_OF (COMMAND_TYPE) ==>

--         L(SUBSTITUTE_CMD) ==> ORIGINAL_PATTERN,LINE1,NEW_PATTERN,LINE2

--         where LINE1 and LINE2 represent a range of lines to which the
--         substitute command is applied.  All occurrences of
--         ORIGINAL_PATTERN are replaced by NEW_PATTERN.


--         { rest of commands specified here}
```

Figure 6. Partial Ada specification of COMMAND_MODULE

BIBLIOGRAPHY

1. Parnas, D. L. A Technique for Software Module Specification with Examples. ACM Communications, vol 15 no. 5, May 1972.

2. Bartussek, Wolfram and David L. Parnas. Using Traces to Write Abstract Specifications for Software Modules. UNC Rep. TR77-012. University of North Carolina, Chapel Hill, NC. 1977.

3. McLean, John. A Formal Method for the Abstract Specification of Software. ACM Journal vol 31 no. 3, pp 600-627. May 1972.

4. Hoffman, Daniel. Trace Specifications of Communications Protocols. IEEE Transactions on Computers. vol c-34, no. 12. Dec 1985.

5. Buhr, R.J.A. System Design with Ada. Prenctice-Hall Inc. Englewood Cliffs, NJ. 1984.

6. Liskov, Barbara and John Guttag. Abstraction and Specification in Program Development. McGraw-Hill Book Co. 1986.

7. Parnas, D. L. On the Criteria to be used in Decomposing Systems into Modules. ACM Communications. Dec 1972.

8. Guttag, John. The Specification and Application to Programming Abstract Data Types. Ph.D. Thesis. University of Toronto. Sept.

9. Latour, L. J. A Programming Environment for Learning SEE a Student's Educational Environment. IEEE 2nd Int'l Conference on Ada Applications and Environments. April 1986.

ADA (TRADE NAME) SOFTWARE ENGINEERING EDUCATION AND
TRAINING SYMPOSIUM (2..(U) ADA JOINT PROGRAM OFFICE
ARLINGTON VA C MCDONALD ET AL. 11 JUN 87

UNCLASSIFIED   F/G 12/5   NL
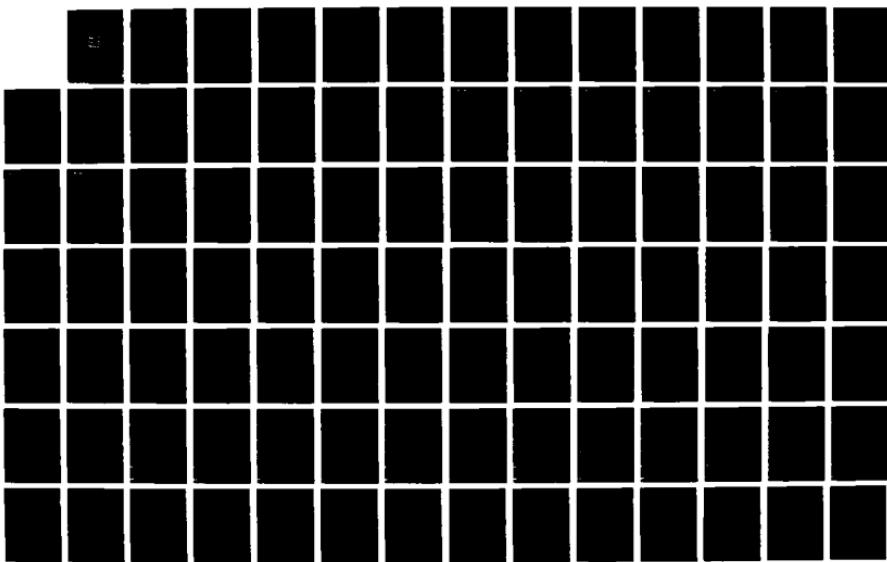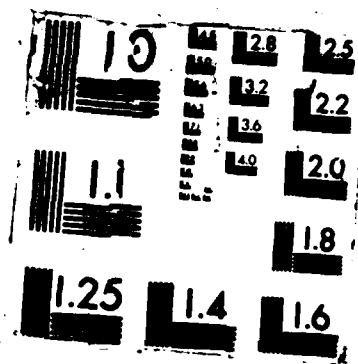
A STUDENT PROJECT TO EXTEND OBJECT-ORIENTED DESIGN

R.F. Vidale
Boston University, Boston, Massachusetts 02215

C.R. Hayden
GTE Government Systems Corporation, Needham, Massachusetts 02194

Abstract This paper describes a project in which five Boston University students designed, in Ada*, an adaptive routing algorithm for a data switching network. The 14-week project used R.A.J. Buhr's object-oriented structured design methodology together with timing diagrams, Petri nets, control skeletons, and Task Sequencing Language to develop the design. By the time coding began, the design had been so thoroughly analyzed that coding, debugging, and testing took only three weeks.

The project demonstrated that engineering students at the senior/graduate level, with one course in Ada and one course in software engineering and using the methodology described below, can implement a multi-task Ada program for an application with high deadlock potential. Key factors in the success of the project were the quality of the students, the design methodology used, the software engineering principles learned through previous project experience, adherence to programming style guidelines, and the Ada language itself.

1. INTRODUCTION

Established design methods for sequential programs, as well as some recent Ada-specific methodologies, do not adequately deal with the added complexity of concurrency. General Dynamics [GEND82] found that structured analysis and design methods, such as Ross and Schoman [ROSS77] and Yourdon and Constantine [YOUR79], were insufficient for real-time systems with tasking. Ruane and Vidale [RUAN84] also found the Abbott/Booch [ABBO83], [BOOC83] object-oriented design methodology inadequate. Buhr's System Design with Ada methodology [BUHR84] addresses the design problems of tasking, utilizing dataflow (cloud) diagrams, structure charts, and cannonical architectures to achieve deadlock-free task interaction. However, there are additional descriptive and analysis techniques, such as timing diagrams [VIDA86], Petri nets [PETE81], and Task Sequencing Language (TSL) [HELM85], which could extend Buhr's approach for achieving desired task interaction. The purpose of this project was to combine these techniques to define an Extended Buhr Design Methodology (EBDM) and use it in a complex multi-tasking application.

---

* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

2. PROJECT OVERVIEW

The GTE Government Systems Corporation, Communication Systems Division (GTE/CSD) defined an Ada design problem to simulate an adaptive routing algorithm for a node within a data switching network. The program required the use of tasking and had a high potential for deadlock. A team of five Boston University students developed a Requirements Specification which was reviewed by GTE/CSD. The students were given programming style guidelines and a preliminary draft of the EBDM.

Before the design of the adaptive routing algorithm was started, the team exercised the preliminary EBDM on a small-scale problem to test the EBDM and to develop a monitor task which would later be used in the network program to monitor and record its execution. The EBDM was revised and served, with some further revisions, to guide the design of the network program. At this point the project was about two weeks behind schedule due to the time spent programming and testing the monitor task. Work then commenced on the design of the network program, which yielded Preliminary Design, Detailed Design, and Software Integration/Testing documents, Source Code, Input Files, Output Files, and a Programmer/User Manual. During the design and testing of the network program, the students kept records of their effort for later analysis of their productivity.

Coding and testing of the program went rapidly and smoothly, largely because of the discipline imposed by the EBDM and the availability of the monitor task. The two-week slippage of schedule was made up and the source code was successfully ported to the DEC/VAX Ada environment at GTE/CSD on May 8, 1986.

3. PROJECT RESOURCES

The project development team consisted of four seniors majoring in Computer Engineering, and a graduate student in Systems Engineering who had a bachelor's degree in Computer Science. All had taken a course in system design using Ada (SC 465), and a first-year graduate course in software engineering (SC 511). SC 465 included a group Ada design project which utilized Ada's tasking features. SC 511 focused more on software engineering, provided a broad coverage of the software life cycle, emphasized software specification and testing, and included a large design project using Pascal.

The host machine for this project was the Data General MV/10000 running the Data General/Rolm Ada Development Environment (ADE), version 2.30, under Data General's Advanced Operating System (AOS/VS), revision 6.03. The ADE included the following set of software tools: text editor, pretty printer, compiler, linker, source code debugger, library manager, text control, and document formatter. All of these tools except the document formatter were used by the design team.

Programming style guidelines for clarity, maintainability, and portability were provided to the design team. A modified McCabe complexity measure, developed by one of the design teams in the SC511 course, was used to specify an upper limit to module complexity.

4. EXTENDED BUHR DESIGN METHODOLOGY (EBDM)

The Extended Buhr Design Methodology used to design the adaptive routing algorithm begins with a definition of the problem, and proceeds through software requirements analysis and software preliminary design. The steps in EBDM which guided the adaptive routing algorithm design were:

1. Identify the objects in the problem, represent the objects by abstract "clouds" on a diagram, and show data flow between the objects. A master cloud diagram will typically evolve out of a series of partial cloud diagrams, each showing a different aspect of the system's function. On multiple copies of the master diagram, threads of control can be shown as chronologically numbered data flows.

2. In conjunction with Step 1, develop scenarios of object interaction using preliminary timing diagrams. These diagrams show data flow, but not the directions of calls. The timing diagrams can portray more than one thread of control per diagram, whereas the cloud diagram cannot.

NOTE: Steps 1 and 2 provide a visual representation of the system, which informally specifies its static and dynamic aspects, without reference to the Ada language.

3. Define global Ada data types for the data flow between objects. Compile the type declarations in a global data types package to check type syntax.

4. In conjunction with Step 3, transform problem-space objects into Ada program units.

NOTE: Steps 3 and 4 transform the system data and objects into Ada data types and program units.

5. Draw a Buhr-style structure graph showing the program architecture. Add Buhr's temporal notations to describe local sequencing of entry calls and accepts. Compile the specifications of the program units to check for interface consistency.

6. In conjunction with Step 5, add directions of calls to the timing diagrams.

NOTE: Steps 5 and 6 establish caller-callee relationships among the program units.

7. Identify task sequencing requirements from the timing diagrams and encode them in Task Sequencing Language specifications. This step forces the designer to specify task sequencing requirements in general terms.

8. Draw Petri nets to describe local and global sequencing of entry calls and accepts. Add timing constraints to the Petri nets.

9. In conjunction with Step 8, write control skeletons for the bodies of the program units. The control skeletons may be parsed for syntatic errors.

10. Walk through the Petri nets to verify the control skeletons.

This completes the specification of the preliminary design.

5. DESIGN OF THE ADAPTIVE ROUTING ALGORITHM

Problem Description: The problem was to implement in Ada an adaptive routing algorithm for a node within a data switching network. Communication between nodes is according to the datagram model [TANE81], in which the network layer accepts messages from the transport layer and attempts to deliver each one as an isolated unit. Messages may arrive out of order, or not at all. A critical requirement of the design is that while user messages are being routed through the network, parallel computing processes at each node are computing the minimum delay times to neighbor nodes, and storing the updated distance table of each neighbor and the updated minimum delay time table of each neighbor. This presents a challenging problem to the designer who must correctly design and implement concurrent, multiple threads of data flow and control.

Requirements Specification: Working from the problem description provided by GTE/CSD, the Requirements Specification was completed on schedule. Since the problem description was expressed in Ada terminology, the Requirements Specification evolved with a strong Ada orientation. Some preliminary design was completed during the development of a Buhr-style diagram presented in the Requirements Specification. When EBDM was applied to the Preliminary Design Phase, the problem was revisited at a more abstract level, using dataflow ("cloud") diagrams, from which revised Buhr-style diagrams were drawn.

Preliminary Design: This phase included the development of the Extended Buhr Design Methodology that would be used in the design the adaptive routing program. The aim of the EBDM is to specify the required task sequencing early in the design and specify its implementation with control skeletons and TSL statements. This specification is progressively refined through a series of steps which model the system from both structural (static) and dynamic points of view. To test the EBDM, the design team wrote an Ada program to simulate the operation of a gas station. The team wrote a monitor task to record the sequence of task interactions in the gas station program so that this tool would be ready for debugging and validating the network program. The preliminary design of the network began a week behind schedule because of time spent developing and testing the monitor, which was used during debugging and testing to monitor task interaction. Extensive design reviews were held during preliminary design, using cloud diagrams, Buhr diagrams, Petri nets, and control skeletons for describing and evaluating the design.

Detailed Design: In this phase, the Ada data structures, program architecture, and control skeletons established during the Preliminary Design Phase were expressed in a Program Design Language (PDL). The PDL included all the constructs of the full Ada language, without any extensions, as found in ANNA, Byron, or TSL, for example. The requirements for task sequencing had already been implemented in the control skeletons developed during preliminary design.

Coding and Testing: By the time coding and testing began, the project was running two weeks behind schedule, due to the time spent developing the monitor and the extra time spent on the preliminary design. This extra up-front effort paid off in the testing phase: only one error was discovered in the entire program during unit testing. Integration testing revealed three coding errors, which were fixed by local code changes. Another error resulted from too short a delay in the task TABLE_READER, which caused it to assume the links were all broken. The problem was corrected by increasing the delay time. During system testing two nodes became deadlocked because each was trying to send a message to the other. The inter-node deadlock problem was resolved by adding a timeout to the SENDER task.

6. EVALUATION OF CODE

The Ada source code was analyzed to determine the number of declarations, statements, and lines of comments. Declaration and statement counts are determined by counting semicolons which act as terminators. The results of the counts are shown below.

Table 6-1

Analysis of Source Code Size

Semicolon Count:

Compilation Unit	Declar- ations	State- ments	Lines of Comments
GLOBALS (spec)	30	0	2
NODE_GLOBALS (spec)	13	0	0
NETWORK	18	1	1
MONITOR_TASK (spec)	12	0	5
MONITOR_TASK (body)	1	0	0
MONITOR	11	46	3
NETWORK_CONTROLLER	8	13	1
NETWORK_MANAGER	1	4	1
NETWORK_STARTER	4	9	0
NETWORK_SHUTDOWN	2	3	0
NETWORK_NODE_TYPE	43	44	4
MESSAGE_ORIGINATOR	10	25	0
TABLE_READER	19	74	69
TABLE_UPDATER	16	46	12
UPDATE	4	19	5
TABLE_MANAGER	13	50	8
NODE_DISPATCHER	5	1	0
RECEIVER	9	46	20
SENDER	13	62	10
TOTALS:	231	443	141

Declarations comprise 34 percent of the total number of statements, or about one declaration for every two executable statements. The high proportion of declarations is characteristic of a strongly typed language, but also indicates that full use was made in this project of the data typing features of Ada.

7. PROGRAMMER PRODUCTIVITY

The students kept a daily record of the hours they spent on the project, which included all effort such as understanding the problem, writing the documentation (Preliminary Design Document, Detailed Design Document, Software Integration/Testing Document, and Programmer/User Manual), coding (Program Design Language representation and Source Code and Input data files), compiling and debugging, and testing.

A summary of the weekly reported effort is shown in Table 7-1 below.

Table 7-1

Hours Worked

Week	Student 1	2	3	4	5	Total
3/2 - 3/8					6.0	6.0
3/9 - 3/15	8.0	3.0	4.0	5.0	6.0	26.0
3/16 - 3/22	5.0	6.0	8.0	5.0	13.0	37.0
3/23 - 3/29	10.0	7.0	9.0	20.0	10.0	56.0
3/30 - 4/5	6.0	7.0	11.0	18.0	11.0	53.0
4/6 - 4/12	6.0	12.0	17.0	7.0	9.0	51.0
4/13 - 4/19	15.0	15.5	15.5	10.0	3.0	59.0
4/20 - 4/26	12.5	12.5	18.5	30.0	11.0	84.5
4/27 - 5/3	13.0	13.5	7.0	11.0	12.0	56.5
5/4 - 5/10	3.0			5.0	10.0	18.0
Totals	78.5	76.5	90.0	111.0	91.0	447.0

The total effort reported by the students was 447.0 hours, for an average of 89.4 hours per student. The lowest value was 12.2 percent below the mean; the highest value was 24.2 percent above the mean. The total effort in programmer-days was 55.88, obtained by dividing the total hours of effort by eight. Based on the analysis of the source code, Section 6, and the above value of total effort, the productivity was 12.1 statements and declarations per programmer-day.

8. PROGRAM IMPLEMENTATION

A test case of the network simulation was run on the Data General MV/10000. In the run, the node was started up, messages were sent through the network, and the network was shut down. Some of the messages were not delivered because the senders in each node timed out when the system could not keep up with the rate at which messages entered the network.

The source code was ported to a VAX/780 running VMS version 4.2 and *DEC ACS version 1.0 at GTE/CSD on May 8, 1986. The overloading of entry names* with enumeration literals was detected by DEC VAX Ada. After the expanded entry names were in place, the code compiled without any errors and without any portability warnings (a feature provided by DEC VAX Ada). On the same day, the code was ported to an ALSYS environment, which required only changing the file names to eight characters. Subsequently, the corrected code was also ported to a VERDIX Ada environment. The only change required was renaming the extensions of the Ada source files from ".ADA" to ".a".

Program executions were run on four different mainframes, summarized below.

Table 8-1

Program Executions

Location	Compiler	Target	Operating System
Boston U.	DG/Rolm	MV/10000	AOS/VS
GTE	DEC VAX	VAX 780	VMS
MITRE	DEC VAX	VAX 780	VMS
MITRE	VERDIX	VAX 8600	UNIX

None of these executions, which included calls to the MONITOR task, experienced any deadlock. The number of messages delivered per execution depended on the machine and the loading by other users of the system. This project did not address performance issues, such as message throughput. Designing for performance is an obvious area for future enhancements of the methodology.

To make certain that calls to the MONITOR task was not a factor in preventing deadlock, the MONITOR was removed from the code at Boston University. A series of executions were run, increasing the rate at messages were injected into the network each time. The rate was increased by decreasing a loop delay statement in the MESSAGE_ORIGINATOR task from an original value of 5.0 seconds down to the effective minimum of SYSTEM.TICK. The only effect was that relatively fewer messages reached their destinations. No deadlock occurred.

95

9. STUDENT ASSESSMENTS OF THE PROJECT

At the completion of the project, the students provided written assessments of their experiences with the EBDM. The major themes of these responses are summarized below.

(1) The students strongly believed that the EBDM forced them to specify and analyze the concurrent threads of control early in the design. They felt that detailed design and testing went rapidly and smoothly because of the understanding acquired during preliminary design.

(2) The deficiencies perceived in EBDM were:

o There was no clearly defined point at which the preliminary design was frozen.

o There was no prescribed means of recording design decisions which would have avoid time-consuming reevaluations of earlier decisions.

o EBDM does not presently cover absolute timing requirements or exception handling.

o TSL is difficult to use.

o Drawing and changing the Petri nets is time consuming.

o TSL cannot handle the use of non-message call sequencing, as in the use of a global flag for shutdown.

o The handling of delays is not presently included in the method.

(3) The techniques most valued in EBDM were

o Cloud (Data-flow) Diagrams

o Buhr Diagrams

o Control Skeletons

Petri nets and TSL were considered less important. The students felt that TSL, if properly mastered and supported by automation, could play a more significant role in design. They acknowledged that TSL motivated a change in design to entry-driven control rather than a data-driven control, which was clearer and easier to code. Timing diagrams were considered to be the least useful of the six techniques. Once the cloud diagrams were annotated to show sequencing, it was felt the timing diagrams were no longer needed. There was a strong desire for automated support of TSL and Petri nets. A requirement to document why decisions were made should be added to the methodology, along with the means to control design freezes. Generally, all aspects of the EBDM could be refined and standardized.

(4) The students unanimously agreed that the most difficult aspect of the project was working with a methodology which was itself evolving as the design process proceeded. Lack of familiarity with TSL and Petri nets (except for one student who had previously worked with Petri nets) also caused difficulty. Understanding the problem was also cited as a difficulty.

(5) Design Changes Forced by the Methodology:

o Using TSL, the design was more expressible if task interaction was based on multiple entries, rather than on data values.

o Petri nets revealed a potential deadlock during shutdown.

o Cloud diagrams revealed early in the design that more efficient operation would result if READ_NEIGHBOR_TABLE were implemented as a task.

(6) The students all felt that time to refine and learn the overall methodology, as well as the individual techniques, detracted from design and testing, especially. Prior training in EBDM would have enabled a smoother application of the methodology.

(7) The students felt the guidelines were easy to use (they had all been exposed to them previously) and were absolutely essential for ease of reading each other's source code.

(8) The testing proceeded more smoothly than expected, which was attributed to EBDM. Errors were more easily discovered because the students were confident the design was correct. One problem with unit testing was defining the correct delays so the stubs would function properly, which slowed some of the unit tests using task stubs.

10. CONCLUSIONS

The project achieved its objective of producing a working design for an intricate asynchronous event-driven system. The EBDM provided the framework in which to visualize, design, and verify the task sequencing requirements before detailed design was begun. The understanding gained through this process was largely responsible for the rapid and successful testing of the program. The success of the project also owed much to the talent and experience of the design team. It was comprised of the best available students who had practiced modern software engineering methods and Ada program development on the host system. They were experienced in top-down modular software development using a team approach. Even with this amount of training, some aspects of EBDM that were new to most of the students. Additional training in the use of Petri nets and TSL would have improved productivity. The students tended to rely on techniques with which they were most familiar or which were quickly learned and easily applied, such as cloud diagrams, Buhr diagrams and control skeletons.

This project demonstrated the effectiveness of the Extended Buhr Design Methodology in guiding the design of a complex multi-tasking application, thereby providing a basis for refinements of the methodology and further extensions in the area of performance issues.

11. ACKNOWLEDGMENTS

12. REFERENCES

[ABBO83] Abbott, R.J., "Program Design by Informal English Description," Communications of the ACM, vol. 26, no. 11, November 1983

[BOOC83] Booch, G., Software Engineering with Ada, Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., May 1983

[BUHR84] Buhr, R.J.A., Systems Design with Ada, Englewood Cliffs, NJ: Prentice-Hall, 1984

[GEND82] General Dynamics, "Ada Capability Study: Design of the Message Switching System AN/TYC-39 Using the Ada Programming Language," NTIS No. AD A123304,5,6,7 (four volumes), November, 1982

[HELM85] Helmbold, D., and D. Luckham, "TSL: Task Sequencing Language," Proc. of the Ada International Conference, Paris, France, May 1985

[PETE81] Peterson, J.L., Petri Net Theory and the Modeling of Systems, Englewood Cliffs, NJ: Prentice-Hall, 1981

[ROSS77] Ross, D.T. and K.E. Schoman, K.E., "Structured Analysis for Requirements Definition," IEEE Transactions of Software Engineering, Vol. SE-3, No. 1, January 1977

[RUAN84] Ruane, M.F. and R.F. Vidale, "Assessing Ada: Implementation of Typical Command and Control Software Functions," Boston University, Boston, MA, August 1984

[TANE81] Tanenbaum, A.S., Computer Networks, Prentice-Hall, Englewood Cliffs, NJ, 1981

[VIDA86] Vidale, R.F, P.A. Szulewski, and J.B. Weiss, "Visualization, Design, and Verification of Ada Tasking Using Timing Diagrams," presented at the First International Conference on Ada Programming Language Applications for the NASA Space Station," University of Houston-Clear Lake, June 2-5, 1986

[VIDA86] Vidale, R.F. "Extending Object-Oriented Ada Design Methodology," GTE Government Systems Corporation, Communication Systems Division, Needham, MA 02194, June 30, 1986

[YOUR79] Yourdon, E., and L. Constantine, Structured Design, Prentice-Hall, Englewood Cliffs, NJ, 1979

An Evolution in Ada Education for Academic Faculty

M. Susan Richman
Ada Education and Software Development Center
Pennsylvania State University at Harrisburg
Middletown, PA 17057

One of the most efficient methods of generating Ada programmers is to teach Ada to faculty so that they may then 'go forth and multiply' by teaching their students and, perhaps, teaching other faculty.

When designing an intensive Ada course for academic faculty, with the objective of preparing them to teach Ada at their home institutions, among the first questions which must be answered are, "How long should the course be? How much of Ada can be covered, with reasonable depth, in one week, two weeks, or a longer period?" The answers to these questions depend in large part on a number of variables over which the instructor may, or may not, have some control.

This paper describes and analyzes the observations of the author based upon participation in three intensive training programs in Ada for university and college faculty --the first as a student, and the last two as instructor. These experiences lead to the conclusion that the most critical factors are:

    (1) the preparedness of the students,
    (2) the quality/speed of the Ada compiler used,
    (3) the extent of the computer support, and
    (4) pre- and post-course assignments.

The importance of each of these factors should not come as a surprise. However, it was gratifying to see that, given sufficient control over each of these factors, virtually all of the language _can_ be covered, with relevant programming assignments, in just one week.

The courses had a significant number of similarities. All three courses were conducted on a college or university campus. The students had heterogeneous backgrounds--from computer science faculty to mathematics, engineering, and business faculty. The primary languages of the faculty were FORTRAN, Pascal, or COBOL. In each course some of the students were campus residents for the duration while others lived within commuting distance. [As a general rule those living at home did not take advantage of evening or weekend lab hours.] In addition to the Reference Manual (ANSI/MIL-STD-1815 A), at least one reference text was used. The lectures generally used overhead transparencies; the students were supplied with paper copies to facilitate note-taking.

Those quantifiable variables observed are summarized in the following Table. Detailed descriptions of the courses and an analysis of the results follow the tabular statement.

	COURSE ONE	COURSE TWO	COURSE THREE
LENGTH OF COURSE	10 weeks	6 weeks	1 week
NUMBER OF INSTRUCTORS	1	2 (1 lecture,1 lab)	1
NUMBER OF STUDENTS	15	13	8
COMPILER	Ada/Ed Interpreter	Ada/Ed Interpreter	Rolm/Data General production quality compiler
EXTENT OF COMPUTER SUPPORT	Limited Other users	Free access Other users	Free Access Other users
PRE-COURSE ASSIGNMENT	Discouraged	None	Reading on the Pascal subset of Ada
POST-COURSE ASSIGNMENT	None	None	Final project
CREDIT VS NON-CREDIT	Non-credit	Non-credit	Credit
FEATURES COVERED IN PROGRAMMING ASSIGNMENTS	All except Tasking Generic units Low-level I/O	All except Low-level I/O	All except Low-Level I/O
DAILY FORMAT	a.m. lecture p.m. lab 9:30 a.m. to 4 p.m.	a.m. lecture p.m. lab 9 a.m. to 4:30 p.m.	Alternate approximately 1 hour lecture/1+ hour lab 8:30 a.m. to 5 p.m.

COURSE ONE

The first course was ten weeks long. There were no prerequisites for the course, other than being faculty in computer science or a related field. Preliminary reading on Ada was discouraged.

One instructor had complete responsibility for the class of fifteen. The class was very loosely structured, meeting from about 9:30 A.M. to 12:00 N and 1:30 P.M. to 4:00 P.M., Monday through Friday. Within that time frame, lectures were generally in the morning with lab in the afternoon.

There was minimal computer support for the first 1 1/2 weeks of the course (two or three students sharing a single terminal for about two hours each afternoon). After the necessary communications were installed, each student had a terminal available throughout the day; evening access was rarely available. The frustrations engendered by this limited access was compounded by

the inefficiency of the compiler, Ada/Ed, and the competition for system resources with other users. Because of the regular users, the system parameters could not be adjusted to optimize Ada/Ed performance. It was not at all unusual to have a compilation job spend one to three hours in the batch queue, before returning information on the latest set of errors. The final project, a programming team simulation of a baseball game (without either tasking or generic units), would not execute with the available system resources. Thus, the inefficiency of the compiler, coupled with limited access to the computer, were probably the greatest factors in limiting the complexity of programs developed.

An extensive list of programming exercises was supplied at the beginning of the course, and students were expected to work through the list, as time and resources permit; they were not expected to turn in their programs for grading. The abundance of exercises focusing on the Pascal subset of Ada may have contributed to the class, as a whole, not doing much programming with the advanced features of Ada.

Three texts were supplied as reference reading. No specific reading assignments were made:

Programming in Ada, J.G.P. Barnes, Addison Wesley
Software Engineering with Ada, G. Booch, Benjamin Cummings
Ada for Experienced Programmers, A.N. Habermann, D.E. Perry, Addison Wesley
Ada An Advanced Introduction, N. Gehani, Prentice-Hall

Guest lecturers [including Grady Booch (Rational), Norman Cohen (SofTech), Maj. Richard Bolz (U.S. Air Force), Georgio Ingargiola (Temple Univ.), and Edmond Schonberg (New York Univ.)] provided insights on specific topics.

No tuition was charged; in fact, the students received a stipend for attending the course. Credit was available upon the taking of a written exam at the end of the course, however, only a few of the students chose this option. Consequently, for most participants any pressure to achieve was strictly through personally set goals.

COURSE TWO

The second course was designed and presented by a team of three faculty who were students in Course One; one was responsible for the lectures, one for the lab, and one for the computer system and the course logistics. A number of changes were made, but we did not have the freedom to make all the changes we might have wished.

Thirteen faculty in computer science or a related field were accepted into the program. Although experience with Pascal was desirable, it was not required. A set of articles on the history of the development of Ada were sent to the participants with the expectation that they would be read prior to the beginning of the course.

This course was six weeks long, as long as the budget would support it. The day was fairly structured, although not rigidly so. For the first three days, lectures began at 9:00 and continued (with breaks) until the material scheduled for that day was covered, usually about 2:00. The remainder of the day was spent on programming. In response to the students' request, this schedule was changed so that both morning and afternoon had lab periods as well as lecture sessions.

The computer used was, as in Course One, a DEC VAX 11/780 with the Ada/Ed compiler. Each student had a terminal, the system was fine-tuned to optimize Ada/Ed performance, and any system problems were dealt with promptly. The excellent system support contributed greatly to the students' programming accomplishments. While compilations could spend quite some time in the job queue, this became a significant problem only with the lengthy programs written toward the end of the course. Interspersing lab periods with lecture periods allowed the student to submit jobs to the batch queue and have the results waiting at the beginning of the next lab period. The lab was frequently open in the evening and on weekends; the primary users of this extra lab time were those students living away from home for the duration of the course.

The lab exercises were designed so that the lecture topics were promptly reinforced with programming assignments. Although the course was four weeks shorter than Course One, the class was able to do more advanced programming assignments: all the features of Ada, except Low Level IO, were covered in these exercises. As in Course One, the final project was a team effort. The assignment was to simulate an air-traffic control for take-offs and landings, incorporating tasking, exceptions, and instantiating a generic queue (developed as a previous assignment) for each runway. Because of time constraints, rather than system constraints, the final project was not completed by any of the teams, although sections of it were completed to the point of being tested.

In addition to the class notes and the Reference Manual, two texts were supplied and Specific readings were assigned to supplement the lectures:

An Introduction to Ada, S.J. Young, John Wiley
Software Engineering with Ada, G. Booch, Benjamin Cummings

A collection of 20-30 Ada texts and related materials were assembled for the use of the class. This enabled the students to examine the Ada texts available for use in their future classes. Additionally, various Computer Aided Instruction courses and the Barnes, Firth, and Ichbiah tapes were there for use and examination in the periods of waiting for a job to work its way through the batch queue. One observation of this experience is that, in an intensive course, it is valuable to have various types of exposure to the concepts. These exposures supplement each other and reinforce concepts that might otherwise be difficult to

absorb.   Another observation is that, no matter which Ada book you read first, the second is always easier to read and, probably for that reason, generally preferred.   The moral of this is, don't allow your students to talk you into changing texts too easily, just because they have read other books that they prefer.
The guest lecturer program was continued with:
Roger Lipsett (Intermetrics) -- Ada as a Hardware Design Language
Grady Booch (Rational) -- Tasking, Generics, and Ada Style
Norman Cohen (SofTech) -- Identifiers, Exceptions, Derived Types
Edmond Schonberg (N.Y.U.)-- Ada/Ed and Ada versus other Languages
Brian Scharr (U.S. Navy, AJPO) -- The role of the AJPO
Genevieve Knight (Hampton Institute)   -- Setting up an Academic
Ada Program
Richard Bolz (U.S. Air Force)   -- Educational Issues with Ada
These presentations were considered a valuable component of the course.
No tuition was charged and the students received a stipend for attending. None of the participants chose the option of receiving credit for the course.   However, all students were expected to hand in their programming assignments and an examina- tion was given to the class. In this way performance objectives were clearly set for the entire class.

COURSE THREE

The third course was designed under major restrictions: the course had to be self-supporting, could be only one week long, and could have only one instructor.   This was not considered to be ideal; nevertheless, the goal was set to accomplish as much as possible within the week.
No specific language was required as background, but Pascal was recommended. In order to optimize the one week of class time (a significant part of which was to be lab time) a pre-course assignment was made: to read those chapters in the primary text that dealt with the Pascal subset of Ada. Unfortunately, this assignment didn't reach the students in time for them to do anything with it. Had they received it earlier, much of the pressure felt by both the instructor and the students might have been relieved. Nevertheless, quite a lot was accomplished in the week.
The day was structured with one-hour lectures interspersed with lab periods at least one hour long. This permitted prompt reinforcement of lecture topics with programming assignments (a new assignment was completed virtually every lab period) and the frequent changes of pace helped alleviate the stress of the highly intensive experience.
Each student had a terminal to a Data General Eclipse MV/10000, equipped with the Rolm Ada compiler.   This was the first of the three courses to have the use of a production qual-

ity compiler; this system went a long way towards making up for the vastly reduced time available for the course. Compilation turnaround time, even for the later more complex programs, was rarely more than five to ten minutes. For smaller programs it was usually one to two minutes. While the system was not dedicated to the Ada students, the other users did not have a significant impact on the performance of the compiler. The lab was available in the evenings and on the weekends (for completion of the final project.)

Each of the topics covered in the lectures (including tasks, generic units, and exceptions) was used in the programming assignments. Because of the time constraints, the later projects were somewhat simplified. For example, a sample specification for a generic queue package was supplied, and the student was expected to complete the package and instantiate it in a main procedure. With more time available, the student would have been required to design the specification as well as complete the package. The time constraint also precluded having a team project. Instead, the student was given one month following the end of the course to complete the final project.

A single text was used in addition to the Reference Manual, <u>An Introduction to Ada</u>, S.J. Young, J. Wiley, and specific reading assignments were made in coordination with the lecture topics. Time did not permit the use of either a classroom library or CAI materials which were available on the system. While a CAI package can be a valuable supplement to an extended course, the fact that it is rather time-consuming makes it's use incompatible with a brief, intensive course. Moreover, the guest lecturer program could not be supported within the time and budgetary constraints.

Unlike the previous two courses, there was a tuition charge (the standard tuition for a three-credit course) and the students were not reimbursed expenses or paid a stipend for attending. The tuition was the same whether the course was audited or taken for credit, and most students chose to receive credit. Even those auditing the course were expected to hand in the completed assignments -- and they did.

RECOMMENDATIONS

<u>Prerequisites</u>: Familiarity with Pascal, or preliminary reading on the Pascal subset of Ada, are highly recommended but not absolutely necessary. Without the students having such knowledge, you run the risk of either losing some of class or boring others.

<u>Number of Instructors</u>: If it is an intensive course, and it is the first time you are teaching it, try to have an assistant. Any class is likely to require adjustments in midstream to meet specific needs of the students and possible system problems. If you are experienced and have <u>all</u> your class materials ready

before the class begins, you will probably be able to cope with minor adjustments as you go along. Otherwise, both you and the class will be under stress which might otherwise be avoided. Longer courses proceed at a rather more relaxed pace so it is less important that the instructor have regular assistance.

System and Compiler: If you have a production quality compiler and good system support, a surprising amount can be accomplished in a short time. Otherwise it probably will require upwards of three weeks to cover the most important language features.

It is critical that educators learn Ada using compilers that support the entire language. A subset compiler that does not support generics, for example, should not even be considered for a course of this nature. It may be appropriate to teach undergraduates a subset of Ada, using a subset compiler, but educators should not learn Ada this way.

Lab exercises: The more intensive the course, the more important it becomes that the exercises be designed to provide experience with the key features of Ada, in synchronization with the lecture topics, and with minimal repetition. Emphasize the reusability and maintainability of Ada code by requiring them to use, modify, and amplify code written for earlier exercises.

Use of texts: The Reference Manual is indispensable, not only for this type of course, but for any Ada course that includes programming. One or more other texts should be used, with assigned readings coordinated with the lectures, to provide at least a slightly different point of view and more detail than the lectures.

Guest lecturers: These contribute significantly and should certainly be included if time and resources permit.

Tuition/Credit: Tuition is a factor that we rarely have control over, but appropriate credit should be offered if possible. In any case, definite performance expectations should be set. Examinations are not necessary, but programming assignments should be required.

Software Engineering: You should be able to expect that your students (they are faculty, after all) have good programming style and follow the principles of software engineering. However, whenever appropriate, the principles of software engineering should be reiterated in the lectures, demonstrated in the classroom examples, and the students should be expected to include them in the design of their programs. If necessary, learning some of Ada's features should be sacrificed in order to concentrate on good design.

CONCLUSIONS

Under the right conditions, i.e.

1. the students are familiar with Pascal or have the opportunity to do preliminary reading on the Pascal subset of Ada
2. the class has good computer system support and the use of a production quality Ada compiler,
3. the days are structured with interspersed lecture periods (50 minutes to 1 hour) and lab periods (1-2 hours)
4. lab exercises are designed to provide prompt reinforcement of lecture topics
5. programming assignments are expected to be completed and turned in to the instructor

you can teach, in as short a time as one week, an Ada course that will be of significant value to the participants. At the end of the week the students will be exhausted, and you may be also, but they will have the foundation necessary for them to teach Ada to their students.

Software Engineering

and its Ramifications to the

Ada® Programming Language

Training Environment

June 11, 1987

Second Annual
Ada Software Engineering Education
and Training (ASEET) Team Symposium

Presented by:

Jerry F. Berlin
Harris Corporation
2101 West Cypress Creek Road
Fort Lauderdale, Florida 33309
(800) 245-6453

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

HARRIS Education Center

ABSTRACT

This paper discusses Software Engineering principles and their ramifications to the success of students in an Ada Programming Language training program. Information on how the Ada Programming Language exemplifies, characterizes, demonstrates and enforces good Software Engineering principles is discussed. Suggestions are presented for the instruction of Ada Programming Language students in Software Engineering principles and methodology along with concrete examples of Software Engineering in Ada Programming Language training.

Background

Harris Corporation is a two-billion dollar a year company that manufactures computers, semiconductors, office equipment, communication equipment and supporting software. The Computer Systems Division manufactures supermini computers used in data processing and real-time applications. One of the software products is the Harris Ada Programming Support Environment, HAPSE®.

HAPSE is an Ada Programming Support Environment (APSE). In addition to the Minimal Ada Programming Support Environment (MAPSE) requirements of a compiler, symbolic debugger, editors, configuration control/configuration management, link loader and job control interface that are described by the Stoneman standard, HAPSE also provides a library manager, optimizer and several extension packages.

The Harris Education Center

The Harris Education Center is a profit center that is responsible for the training of both customers and employees. Our customers come to us from the government, industry and educational institutions. They employ Harris computers in office automation, engineering, data processing and real-time applications. Students who are Harris employees range from new hires to experienced analysts, technical writers, programmers and engineers.

The courses that we offer cover all aspects of the hardware and software for Harris computers and their components. The majority of training takes place at our Fort Lauderdale, Florida, Education Center; however, for large groups, the training can take place on-site, at the customer's location.

The Ada Programming Language Curriculum

All of the Harris Education Center's courses are developed by its staff of instructors. Throughout the development of a course, comments and suggestions are solicited from groups in product development, marketing and the field offices to insure that the course is designed to satisfy as many of our customers' requirements as is possible.

The Ada Programming Language curriculum is designed to instruct programmers in the Ada Programming Language and Software Engineering. The Ada Programming Language curriculum is comprised of four courses. The recommended sequence for the courses is:

® HAPSE is a trademark of Harris Corporation.

1. *Introduction to the Ada Programming Language* (5 days): Software Engineering principles and the Ada Programming Language are introduced.

2. *Advanced Ada Programming Language* (5 days): Advanced features and topics of the Ada Programming Language are presented, and how Ada enforces good Software Engineering is discussed.

3. *Harris Ada Programming Support Environment* (3 days): All HAPSE tools, utilities, implementation dependent details and packages are covered.

4. *Ada Programming Language Workshop* (5 days): Program maintenance, life-cycle support and applications are taught.

Description of Software Engineering

Software Engineering methodology is a process that can be implemented in varying degrees with virtually all programming languages. For maximum effectiveness, however, it should be implemented with a programming language such as Ada, that provides all of the necessary features and abilities required to properly utilize Software Engineering.

Software Engineering principles include the concepts of: modularity, abstraction, information hiding, localization, uniformity, completeness and confirmability.

Abstraction keeps the underlying details of a program unit away from the programmer. All that should be known about any program unit is the means to interface with it.

Information hiding requires that the underlying details be made private and inaccessible from higher levels of the abstraction.

Modularity is the top-down decomposition of program function into small, easy to maintain, discrete program units or modules. Ada subprograms and packages are designed with this this concept in mind.

Localization goes hand-in-hand with modularity. When programs units are organized into discrete modules, local parameters in each module can provide a greater degree of independence. These local parameters will not interfere with or change the meanings of values in other parts of a program, nor can any other part of the program interfere with the local variables.

Uniformity requires all program units to be written in the same, consistent style.

Completeness ensures that all important items of a program unit are accessible.

Comfirmability allows a program unit to be decomposed for testing.

All of these Software Engineering methodology concepts, when followed, enable

programmers to update and modify programs previously written by themselves or others in a timely manner with a minimum of expense.

Meeting the Needs of the Students

The degree of enlightenment among organizations on Software Engineering methodology varies significantly. Most organizations are familiar with and enforce good Software Engineering practices. However, there are organizations that either are not acquainted with Software Engineering or do not encourage its use. This variation among organizations also extends to their employees. Not all employees are familiar with Software Engineering methodology. Fortunately, some employees have had courses while attending college or while on the job that have exposed them to Software Engineering methodology; however not all students have had this opportunity.

Many students do not care for a theoretical approach while learning Software Engineering, but are more interested in obtaining practical information. They have projects to complete, code to produce and deadlines to meet. Frequently, they have specific project requirements and problems that go beyond the scope of the objectives of the course. Furthermore, each organization has its own reasons and needs for sending their people to a course, and on occasion, these can deviate from the published course description and objectives.

Each student has his own method of learning. Some students learn best by listening to the lectures while viewing the associated visual aids. Others experience their best learning when reading through the manuals and books that are given to each student as part of the course. Still, other students gain the most from the practical experience provided by the student exercise assignments. It is the instructor's responsibility to be aware of the various methods that students employ to learn the material and to provide an environment that enables all students to have their best opportunity to learn.

The approach that we take closely integrates the three methods of learning. We provide a Student Guide containing the lecture notes for the appropriate course, the *Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A) and *Software Engineering with Ada* by Grady Booch. After presenting an overview of Software Engineering, we provide practical and complete example programs that stress good Software Engineering. All Software Engineering is integrated with Ada syntax and examples. Since students are not equally capable of understanding a top-down approach for learning Software Engineering or the syntax of Ada, some explanations, descriptions and examples are started on the component level and then build up to tie the components together. All lectures must be long enough and detailed enough to be

complete, yet not so long as to allow boredom to set in.

Lecture and Theory

The student guide is a tool for learning the Ada Programming Language syntax. Additionally, example programs are provided. These are complete programs rather than a segment or a portion of a program. They are brief, clear and to the point. All example programs follow good Software Engineering methodology and in many cases, build on one another.

One lecture is dedicated to the usage of the *Reference Manual for the Ada Programming Language.* The students are taken through it section-by-section. The table of contents, annex, appendix, index, syntax diagrams and cross reference are explained and the section and paragraph numbering is stressed. During the lectures, the *Reference Manual for the Ada Programming Language* is used to provide extra and optional information and to supply complete answers to questions that come up during the class.

Laboratory Orientation

Before starting on the first laboratory exercise, students are provided with an orientation to the computer system. Not all students are familiar with Harris computers. Since the Ada Programming Language courses are designed to be portable, all system information is provided in a separate, short lecture.

During this orientation lecture, students are provided with information on basic operating system job control, compiler, linker and editor commands. Students learn how to invoke the compiler with an option for error processing to make their debugging of programming assignments easier. This error processing provides feedback to the students that enchances the learning process. When an error in a student's source code is detected by the compiler, the appropriate error message is embedded in the source code file and the editor is then called. The majority of the embedded error messages cross reference to the *Reference Manual for the Ada Programming Language.* This error processing feature allows the students to get immediate feedback to assist them in learning from their mistakes. When all errors are corrected and the file is recompiled, the student can then link and execute the program.

For students that do not have the need or the desire to learn the Harris job control for HAPSE, there is a "compile" script to assist them in the compilation, error handling, editing and linking process. The script allows these students to first concentrate on Ada programming without the inconvenience of an "information overload". They can

112

learn Ada first and then, later on, they can learn the details of job control.

Student Exercises and Practical Applications

Just as it is important for students to have the opportunity to learn new concepts, they must also have the opportunity to reinforce their knowledge. A sufficient amount of time is built into the course schedule for students to learn and review the information. Students are given enough time to solve problems properly without the pressure of having to rush to finish. Students have time to "play" with the their newly learned information, to improve their programs, and to experiment with "what if" scenarios. For the more proficient, faster students, optional work and ideas to research are made available.

The first programming assignment provides students with an opportunity to be accustomed to the editor and compiler. Students are provided with a short, poorly designed program to debug and correct. The program contains a bubble sort algorithm which is reviewed in class before starting the assignment. This program requires no input, and when successful, instant feedback is provided through the output of a properly sorted list. This program is then used in virtually all of the following exercises. In each student exercise, something from a program written in the previous student exercise is modified. This continual reuse of the previous program simulates the life cycle of a software project. Students are encouraged to follow Software Engineering methodology to allow them to make more effective use of laboratory time.

After the students get the bugs and logical errors out of their program, the next step is to improve the code. The original program as provided, is intentionally written using very bad programming style. Students must improve the program by implementing Software Engineering methodology. The students must improve the loop control structures and institute the use attributes.

The next step is to make the program interactive. Also, students modify the type definitions of the items to be sorted from the predefined integer type to records that contain a string type and an integer type.

In the next assignment, students modify the program to improve the modularity with subprograms. The sort program's comparison test code used to determine the ordering of the records and the code to swap records are both placed into subprograms. In addition to these two subprograms, an overloaded Get_line procedure and an overloaded Put_line procedure are created by the students to handle the input and the output of the records.

The next modification the students make to their sort program is to place the four subprograms from the previous assignment into a package. Students should soon realize that they now have software that can be reused on future assignments.

In next modification of their sort program, the students write and instantiate generics. The modifications to the sort program now take two separate branches. First, the students modify the sort program from their "subprogram" assignment to be generic subprograms. Second, the package from their "package" assignment is modified to be a generic package.

The students now modify the program to use Text_IO for file manipulation. Input is from a file and output is either directly to the printer or to a file.

In the final modification of their sort program, students change the code in the bubble sort algorithm and implement the Direct_IO package. Rather than storing the records in an array in memory, records are stored in a disk work file. The program must be modified to directly read from and write to records on disk. Students are encouraged to use the packages that they wrote in earlier lab exercises.

For variety, to encourage creativity and to provide for the more capable students, several additional program assignments are provided throughout the course. These include programs on exception handling, the Calendar package and access types.

The exception handling assignment is a short program written early in the week requiring the use of raise statement and user defined exceptions.

The Calendar package is used in a "speed typing test" program. Students write a program to display a line of text and to then time how long it takes for the user to type in an exact duplicate of that line. The Calendar package supplies the timing functions.

The access types program is supplied to the students in a partially written form. The program prompts for and reads in records comprised of strings and integers. These are then added to a list. Students must write a procedure to print out a list of all records entered.

In the final exercise, students write a program to calculate the Fibonacci number sequence. This is an open assignment where the students use whatever they require from Software Engineering and the Ada Programming Language. If the class is running short on time then this assignment is optional.

The students are constantly encouraged to use the Software Engineering principles that are being taught in conjunction with the Ada Programming Language concepts. By having programs build on one another, an emphasis is placed on good Software Engineering. Students that abide by these principles accomplish their projects in a more efficient manner.

Throughout the entire course, during both the lecture and the lab exercises, the advantages of Software Engineering are constantly being taught, discussed,

encouraged and utilized. During the lecture, when each new topic is introduced, the appropriate Software Engineering principles are discussed and reviewed. Throughout the student lab exercises, alternate solutions reflecting the Software Engineering principles are suggested by the instructor.

Conclusion and Summary

Each student has his own method of learning and must be assisted through the educational process accordingly. Students cannot be rushed; they need time to learn and to become comfortable with their new knowledge. The course and the instructor must be flexible to accommodate the widely varying learning styles of all students.

Each organization's motivation for sending their employees for training is also unique. Students attend class not only to learn but to get help and advice. Practical information is just as important as learning the theory of Software Engineering and the syntax of the Ada Programming Language.

The majority of the cost in a program's life cycle is in the modification of the program after it is originally written. Any Software Engineering training course should simulate real world, practical situations as accurately as possible.

Software Engineering is a process, while the Ada Programming Language provides results. The Ada Programming Language was designed to readily support Software Engineering principles.

A properly designed training course for Software Engineering principles and the Ada Programming Language must integrate the information so it can be presented in a relevant and realistic way.

BIBLIOGRAPHY

Booch, G. *Software Engineering With Ada, Second Edition.* Menlo Park: Benjamin/Cummings Publishing Company, Inc., 1987.

Reference Manual for the Ada Programming Language, Ada Joint Program Office, Department of Defense, Washington, D.C., February 1983, ANSI/MIL-STD-1815A.

ADA* TRAINING:

A DEVELOPMENT TEAM'S PERSPECTIVE

BY

R.J. VERNIK

Prepared for

Second Annual Ada Software
Engineering And Training (ASEET) Team Symposium
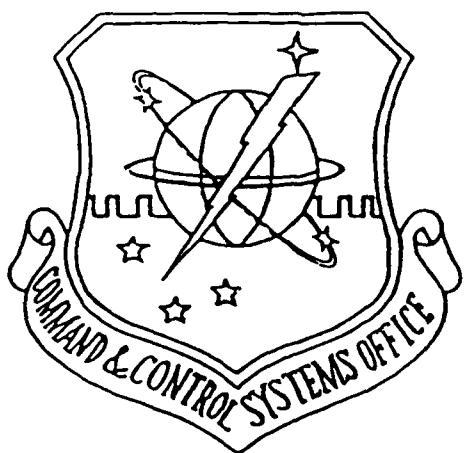
Prepared at
Standard Automated Remote to AUTODIN Host (SARAH) Branch
COMMAND AND CONTROL SYSTEMS OFFICE (CCSO)
Tinker Air Force Base
Oklahoma City, OK 73145 - 6340
COMMERCIAL (405) 734-2457 / 5152
AUTOVON 884 - 2457 / 5152

* Ada is a registered trademark of the U.S. Government
(Ada Joint Program Office)

ABSTRACT

This paper addresses training requirements for
Ada development teams as seen from a development
teams perspective. As such, this paper provides
a more practical view of Ada training than is
often proposed by educational institutions.

The first section covers introductory material.
Some background information is provided on the
scope of the paper and the basis for evaluating
training requirements. In addition, this section
covers the purpose of the paper and outlines some
assumptions and constraints.

The second section describes Ada training needs.
This section argues that to obtain the full
benefits of Ada related technology, the training
program should include training in software
engineering, development methodologies, support
environments, and language syntax. The necessity
to train managers is also discussed.

The third section describes training experiences.
This section is based largely on the Standard
Automated Remote to AUTODIN Host (SARAH) project
training program. As such, it should provide
practical information to organizations who are
contemplating developing Ada based software
systems.

The last section summarizes the major points and
makes recommendations on a possible approach to
Ada training.

ADA TRAINING: A DEVELOPMENT TEAM'S PERSPECTIVE

1. INTRODUCTION

1.1. BACKGROUND

Software development using the Ada language and associated software engineering technology requires a higher degree of training than was required for older languages. There have been many articles outlining the benefits of using the Ada language [1,3,4]. Through Ada and associated software engineering technology we can gain significant cost benefits through code reuse, transportability, lower maintenance costs, and increased productivity. Managers are quick to point out these benefits, yet many do not understand that without the correct engineering approach, extensive use of tools, and a high level of managerial support, these gains will not be achieved. If development teams are to successfully develop Ada software in this very complex environment, they must receive training in a number of areas. In addition to training in Ada syntax, team members require training in software engineering, development methodologies, and programming support environments.

So that potential Ada developers could gain a practical insight into what was required to successfully develop Ada software, the Air Staff tasked the Command and Control Systems Office (CCSO) with evaluating the Ada language while developing real-time digital communications software. The evaluation reports were to consist of a number of papers, one of which was to deal with training requirements. This paper is derived from the original paper on training.

CCSO chose the Standard Automated Remote to Automatic Digital Network (AUTODIN) Host (SARAH) project as the basis for this evaluation. SARAH is a small to medium size project (approx. 40,000 lines of source code) which will function as a standard intelligent terminal for AUTODIN users and will be used to help eliminate punched cards as a transmit/receive medium [8]. The development and evaluation environment for SARAH consists of a Digital Equipment Corporation VAX 11/780 which hosts the SOFTECH Ada Language System (ALS), several IBM PC-ATs which host the ALSYS Ada compiler, a Burroughs XE550 Megaframe, and several IBM compatible PC-XT microcomputers. The SARAH software targets are the IBM compatible PC-AT and PC-XT microcomputers. Since the PC environments do not support many of the Ada Programming Support Environment (APSE) tools required to maintain a stable software baseline (such as configuration management), the ALS was to provide these features.

The SARAH team required training in several areas. These included training in software engineering, the Ada language, the latest design methodologies, the ALS environment, and staff

management and analyst training. Since none of the team members had previous Ada development experience, management was quick to realize that the training overhead would be high. Indeed, the amount of training procured for the SARAH project was far in excess of that procured for projects developed in other languages.

Several methods were used to provide the required training. Formal training was obtained from both commercial and government sources. In-house training was conducted using a Computer Aided Instruction (CAI) package, instructional video tapes and lectures. The SARAH team also gained a great deal of practical knowledge through their involvement in the local and national Ada communities.

1.2. PURPOSE

The aims of this paper are to:

.   Outline training needs for Ada software development teams.

.   Provide practical feedback on Ada training to prospective Ada developers.

.   Make recommendations on a possible approach to Ada training.

1.3. ASSUMPTIONS AND CONSTRAINTS

The assumptions and constraints are as follows:

.   A major constraint is the size of the SARAH project. Since the SARAH project team is small (14 persons), some of the experiences reported in this paper may not be appropriate for training larger development teams.

.   The SARAH team members had a variety of previous experience. Some members had very little software experience, others were well versed in assembly language programming, and some were experienced Pascal programmers.

2. ADA TRAINING NEEDS

This section outlines Ada training needs as seen from a development teams perspective. Ada development teams must be properly prepared if they are to design and develop high quality Ada software. Experience with the SARAH project has shown that development teams must be trained in modern software engineering practices, be able to select and design with a methodology that is suited to Ada, use environment tools to increase productivity and maintain a stable software baseline, and have a good command of the Ada language. In addition, if the project is to be successful, then managers must have a good understanding of what is required to produce good quality Ada software within budget and time constraints.

2.1. SOFTWARE ENGINEERING TRAINING

Software engineering must be stressed if software development using the Ada language is to be successful. The Ada language was designed by software engineers who based their design on modern software engineering principles [1]. Ada supports many of the features of modern software engineering. For example, Ada provides facilities for structured programming, strong data typing, separate compilation, information hiding, data abstraction, and procedural abstraction. These facilities, when properly applied by designers and programmers, can reduce maintenance costs, promote transportability, and improve reliability and survivability. These benefits are realized only when those facilities are used in the manner intended by its designers, otherwise the major benefits of the language will be lost.

Since many of Ada's features relate directly to modern software engineering, the language is easier to learn if it is presented in a manner that facilitates the implementation of these engineering principles and goals. The size of the Ada language has often been criticized; some authors have indicated that it would be beyond most programmers to ever gain a working knowledge of the language. Many of these comments were made in a comparison of Ada to traditional languages such as FORTRAN and COBOL. The comments are well founded if the same approach to teaching these second generation languages is used to teach Ada. Unlike traditional high order languages, Ada provides the engineer with language capabilities built in to facilitate the solution of a complex array of problems not available in other languages. Educators need to introduce each of these capabilities and explain their purpose. For example, an apprentice builder must be told that a saw is used for cutting wood and then shown the correct way to use the saw. Similarly, an Ada software engineer must be introduced to the concept of the package and then shown how this can be used for data abstraction. Students should understand that the Ada language itself does not solve

problems. They should learn the purpose of each Ada engineering
tool and how each can be used to successfully develop software
systems. If Ada is taught in this way, students will more easily
remember the syntax of the language because of its relationship
to software system engineering.

In addition to introducing students to the facilities of
software engineering, software engineering training must cover
many other aspects of software development. For example,
instruction should be given on software maintenance,
configuration management, documentation, testing, software reuse
and the use of programming support tools. As discussed, through
software engineering we can effectively teach Ada syntax;
however, the success of a project is determined not by the code
alone. The scope of software training should cover the full
software lifecycle. The student must be made aware of the impact
that a bad design will have on software maintenance. Similarly,
if the configuration of a software product is not carefully
controlled, the student should realize that the software could be
made useless. If students are taught fundamental software
engineering principles and techniques, organizations can gain
significant long and short term cost savings. These gains will be
realized because productivity is enhanced by software reuse and
reduced maintenance efforts, software systems will be more
easily transportable to different hardware, and software
engineering-Ada trained personnel will quickly become productive
when transferred to new Ada projects due to standardization of
the software development process.

2.2. DEVELOPMENT METHODOLOGY TRAINING

Development methodology training should cover the different
analysis/design methodologies as well as the use of program
design languages (PDLs) [9]. There are many methodologies
available for software development. Those most applicable for the
development of Ada software are covered in the Methodman I [10]
documents. Since no one methodology currently covers the full
software lifecycle or all design paradigms, students should be
introduced to a number of the methodologies so that they can
select the features of each which would best suit their needs.
For example, Object Oriented Design (OOD) [12] provides a
powerful method for design but does not effectively address the
analysis phase. Also, OOD is largely ineffective for the design
of real-time process driven systems. In these cases a methodology
which supports process abstraction is required and so the
designer may chose a methodology such as Jackson System
Development (JSD) [13] or the Process Abstraction Method for
Embedded Large Applications (PAMELA) [14]. Some cases may call
for a multi-paradigm approach [15] which requires the inclusion
of the concepts of two or more methodologies. Clearly, courses
teaching a single methodology do not prepare students for the
variety of design and development problems that exist.

Software engineers must be able to stay current with advances in software development technology. The next few years will bring many new methodologies, many of which will be superior to those used today. Students should be introduced to the different design methods such as procedural, type and process abstraction so that they have the background for understanding the scope of these new methodologies. In addition, new methodologies will be developed which will more effectively cover the testing, integration and maintenance phases of software development. The limitations of existing methodologies should be highlighted so that future software engineers can more readily identify methodologies which will support full lifecycle needs.

Educators should cover the arguments governing the use of PDLs [9] and provide an introduction to the types of PDL currently in use. Students should be able to determine whether a PDL is required and if so what type of PDL would best suit their needs. The whole area of PDL is being hotly debated. There are some who believe that a PDL is not required for Ada development. They suggest that the language itself can be used as a PDL. Those arguing for PDLs are in disagreement over what form it should take. If an organization elects to use a PDL, or a contract specifies that a PDL will be used, the training requirement should be assessed and the training budget should be adjusted appropriately [7].

2.3. SUPPORT ENVIRONMENT TRAINING

Training in the use of Ada Programming Support Environment (APSE) tools is required if an organization is to achieve productivity benefits. Software engineers cannot be expected to fully utilize these environments unless they are trained in their use. For example, the Stoneman model proposes the use of many complex tools for the development of Ada products 16]. The tools provide functions such as configuration management, symbolic debugging, frequency and timing analysis, and code formatting. This type of environment is very complex and users require a high level of training if they are to use the tools effectively.

Students should first be given an overview of the environment. The overview should provide an introduction to the command language, database organization and file administration. In addition, the overview should cover basic operations such as invoking tools, compiling programs, exporting, and linking. After the students are familiar with these basic operations they should be given a more advanced user course which covers the tools in detail. Educators should introduce each tool, demonstrate its use and show how the tool could be used to improve quality and productivity. There is little doubt that significant productivity gains can be made through the use of automated tools; however, managers must understand that training is required if these tools are to be used effectively.

APSE administrator training will be required for effective use of the overall APSE system. The APSE administrator position on a development team is enormously important. The administrator is responsible for controlling access, transmission/reception of reusable library modules, providing incremental updates and database administration/maintenance. If these tasks are not done properly, the whole development effort could be placed in jeopardy. APSE administrator training is therefore an important part of an overall training program for Ada development teams.

2.4. LANGUAGE TRAINING

Language features should be taught in conjunction with software engineering principles and goals [9]. As discussed earlier, Ada is the product of software engineering and supports many modern software engineering features. If the Ada language is broken up into the logical partitions which correspond to engineering building blocks, students will find it easier to understand and use the entire language. Moreover, the language was designed to be used in conjunction with development methodologies and programming environments [2]. Students should be introduced to these areas prior to being submitted to language training so that they can understand how different language features apply to the overall development process. If the full benefits of Ada are to be achieved then language training cannot be conducted in isolation.

The curriculum for Ada language training can quite effectively be divided into two separate areas: basic and advanced topics. After completing a basic course, the student should be able to:

- Declare and use Ada objects and types,
- Understand and formulate Ada statements,
- Code and call Ada subprograms,
- Design and use Ada packages,
- Raise and handle exceptions,
- Perform input/output.

The basic course should give the student a working knowledge of sequential Ada.

An advanced Ada course should cover the concurrent aspects of Ada and low level features. The course should stress important design features such as the use of generics for software reuse. Upon completion, students should be able to design, code, and test Ada programs that use generics, low level features, and tasks. Moreover, the course should allow students to apply sound software engineering principles to produce well-designed Ada systems.

The length of time required for formal Ada language training is dependent on the students' previous experience, the amount of

pre-course preparation, and the requirement for practical training. Several organizations (e.g. Softech Inc. and the US Air Force Air Training Command (ATC)) suggest that at least six weeks are required if this type of training is to be successful. These courses include a high degree of practical training and assume no previous experience in structured programming.

There are other training organizations that believe that the Ada language can successfully be covered with two 40 hour courses; however, several considerations need to be made if this training is to be effective. First, students should have at least some knowledge of the Ada language prior to commencing formal training. This could be achieved through the use of a CAI package, video tapes and self-study. Second, since a large amount of material needs to be covered in a relatively short time, practical training should be limited and structured towards reinforcing the major concepts. Third, consolidation time of at least one week should follow each course so that students can have the opportunity to consolidate their knowledge on practical exercises.

Student assessment should be provided by the training organization. These assessments are beneficial for a number of reasons. First, students are generally more motivated towards retaining information if they know that they will be tested at the end of a training period. They tend to compete more with their fellow students and do not like to see unfavorable reports sent to supervisors. Second, through student assessments, managers are given an insight into the effectiveness of the training program by monitoring student progress. Managers should be provided with a copy of the assessment scheme and results so that they can use the information to more effectively manage their software projects.

In summary, formal Ada language training could be covered in as little as 80 hours; however, the training must be intense, practical training must be well organized, time must be allocated for consolidation, and the students must be very motivated.

2.5. MANAGEMENT TRAINING

Software development using the Ada language and associated software engineering technology will only be successful if full support is provided by management. To do this, managers must have a firm understanding of the technology being used and they must be introduced to some of the problems that may be encountered during development. Managers at all levels within an organization should receive Ada technology training. The initial investment for Ada development is high and there are a number of potential pitfalls. Managers must be educated in this new technology so that they will have the ability to support development teams when problems arise. As with any new technology, there are many problems yet to be overcome and the organization will benefit

only if management works together with development teams to solve these problems.

Managers should be introduced to the Ada community and provided with information on where they can find additional information and help. There are many sources of Ada related information available to managers. For example, the Ada Joint Program Office (AJPO) operates the Ada Information Clearinghouse which distributes Ada related information. Moreover, various organizations have been formed to act as forums for Ada discussion (e.g. SIGAda and AdaJUG). Management training should cover this type of information so that managers can keep current with new advances in Ada technology.

The major benefits of Ada based technology should be covered so that the manager understands what can and should be achievable. Managers should be shown how the Ada language and associated software technology can lower maintenance costs, improve software transportability, and improve reliability. Software reuse should be covered. Managers should be shown that by designing software with reusability in mind, significant cost savings can be made. In addition, managers should be aware that libraries of reusable software such as SIMTEL-20 [18] now exist. If managers are made aware of how Ada based technology can be used to develop quality and cost effective software products, they will be in a better position to help introduce this new technology into their organization.

Managers should be informed that with Ada there is a high initial investment and that some of the benefits will only be realized in the long term. The capital investment is significantly different from what was required for older languages. For example, to obtain many of the productivity benefits associated with Ada, automated tools are required. Software maintenance cost will only be reduced if a proper development methodology is applied and so this translates into higher training costs. In addition, the length and cost of language training will be higher than for older languages. Software personnel will be more highly trained and so key personnel will most probably be more expensive to hire. The manager must be shown that the cost savings through less retraining, higher productivity and less maintenance will far outweigh the high initial investment.

3. TRAINING EXPERIENCES

This section discusses the Ada training experiences of the SARAH project. The SARAH team received both formal and in-house training which covered a broad range of topics required for Ada development.

3.1. FORMAL TRAINING

Members of the SARAH team received the following formal training:

- **Development Methodology Training:** Basic training in development methodologies was provided by EVB Software Engineering. In addition, team members attended a number of tutorials on Ada design at local and national SIGAda conferences.

- **Language Training:** Language training was provided by Intellimac Inc, Rockville MD. Two courses (each of 40 hours) were provided and covered basic and advanced language features.

- **Management Training:** Management training was provided by the U.S.A.F Air Training Command (ATC). The course was one week in duration.

- **Environment Training:** The SARAH team received three courses on the SOFTECH ALS. The courses consisted of a two day introduction, a one week users' course, and a three day administrators' course.

No formal software engineering training was provided for the SARAH team. However, the chief designers all held professional computer science and engineering qualifications. In addition, they were well versed in modern software engineering practices through their participation in advanced workshops and tutorials. Many of these tutorials dealt with how Ada can be used to satisfy the principles and goals of modern software engineering.

3.1.1 Selection

Training Sources. One of the best sources for providing details of currently available Ada training is the Catalog of Resources for Education in Ada and Software Engineering (CREASE). This publication is available for distribution through the Defense Technical Information Center (DTIC) and the National Technical Information Service (NTIS). The accession number for this document is AD A156 687. Further information on CREASE can be obtained by contacting the Ada Information Clearinghouse (AdaIC). In addition to CREASE, the AdaIC provides training

information in their periodic newsletters.

Research. CREASE can provide information on training courses; however, managers should do additional research to determine whether a particular course will be applicable for their project. One of the best ways of achieving this is to talk to others who have recently undergone training. The national and local SIGAda conferences are a good place to do this type of research. Apart from being able to discuss training needs with the many experienced people who attend, many of the training organizations are available to discuss their training curriculums.

Specify Training Needs Precisely. Since there is a large range of Ada training currently available, organizations need to correctly specify their training requirements or the training received may not cover training needs. For example, when outlining the requirement for practical training, the specification should indicate that the training must be conducted with a validated Ada compiler. Several training organizations are currently using JANUS Ada compiler hosted on IBM-PCs for practical training. The cost of this training is generally lower than comparable training using validated compilers. However, since JANUS does not provide the advanced features of the Ada language, the practical exercises are limited to basic features. An organization considering vendor training should research the market well and provide a precise specification of their needs.

3.1.2 Some Problems

High Training Costs. Ada training can be expensive. The SARAH development team received development methodology, language, and environment training through commercial sources. Acceptable quotes for 80 hours of Ada language training for 20 personnel ranged from $20,000 to $77,000. For this amount the vendor was required to provide equipment in-house for the practical sessions. To obtain training in current design methodologies, CCSO sent personnel to the vendor's site. The training cost was $1,125 per week for each student. Travel and lodging costs increases this amount considerably. CCSO found that it was far more economical to have the vendor train in-house if more than six personnel required training. However, if the training is conducted in-house, it must be segregated from the work environment, otherwise the training program could be severely jeopardized.

Specify Training Needs Precisely. Since there is a large range of Ada training currently available, organizations need to correctly specify their training requirements or the training received may not cover training needs. For example, when outlining the requirement for practical training, the specification should indicate that the training must be conducted with a validated Ada compiler. Several training organizations are

currently using JANUS Ada compiler hosted on IBM-PCs for practical training. The cost of this training is generally lower than comparable training using validated compilers. However, since JANUS does not provide the advanced features of the Ada language, the practical exercises are limited to basic features. An organization considering vendor training should research the market well and provide a precise specification of their needs.

Evaluation. The lack of student and instructor evaluation is also a problem. Most training organizations will not volunteer to administer student tests and provide results. As previously described, there are definite benefits to providing some form of assessment. If student assessment is required, then this must be stated in the training requirements.

Procurement Problems. For government agencies there is a long lead time for procurement. This must be taken into account in the project schedule. CCSO found that it took seven months to procure training. No doubt this time could be shortened if the requirement received higher priority; however, there are fixed lead times associated with competitive acquisition. Managers must take this into account, otherwise the lack of training could severely affect the development schedule.

3.2. IN-HOUSE TRAINING

The SARAH team received in-house training through informal in-house lectures, Computer Aided Instruction (CAI), video tapes, and self-study. The relative merits of each approach, and the problems that were encountered are covered in the following paragraphs.

3.2.1 Informal In-house Lectures

Informal in-house lectures can be effective if Ada experience already exists in the development team and the organization has a large number of personnel to be trained. CCSO attempted to establish an in-house informal lecture program for the SARAH project team but found that it was not cost effective. The main reasons for establishing this type of program were to:

- provide team members with a good insight into the Ada language prior to formal training.

- provide a means of Ada technology transfer between the SARAH project and other branches at CCSO.

The main reason for failure was that CCSO did not have personnel available with sufficient Ada experience to conduct this type of lecture program. Since the amount of time used for preparation adversely affected team productivity, the SARAH managers canceled the in-house lectures and placed more emphasis on self-study, use of the CAI package, and the viewing of video tapes.

Organizations should be careful in establishing an in-house lecture program if there is insufficient Ada and software engineering experience available. During an early experience with Ada, CCSO developed this type of program to train a team involved in evaluating the Ada language for use in digital communications applications. The personnel involved in developing the course had never received formal Ada training and based their instruction on traditional language technology. As such, the Ada software produced by the team resembled FORTRAN code. Since the team did not use any of the advanced features such as packages, generics, or tasks, the software was unstructured and difficult to understand. Moreover, the benefits of using the Ada language were not recognized since the team did not apply modern software engineering practices.

3.2.2 Computer Aided Instruction

A CAI package is very beneficial for teaching Ada syntax and for consolidation during and after formal Ada language training.

The SARAH team used the ALSYS "Lessons on Ada" CAI package. This package provided a high level of training in Ada syntax. The extensive use of examples and problems make this package very effective. The interactive nature of the package also allows users to review specific areas and so it serves as a good reference source. IBM-PC compatible microcomputers were used to host the package. The package was used extensively prior to formal Ada language training so that the students could gain maximum benefit from the instructor's experience. Since they were already familiar with much of the Ada syntax, they were able to concentrate on how the language could be used to develop the SARAH system. The ALSYS CAI package has been used throughout the SARAH project to allow software personnel to revise certain areas of the language.

The CAI package allows for training flexibility; however, usage must be controlled so that all personnel benefit from the package. Members of the SARAH team were able to organize training to suit their other work commitments. Enthusiasm for the package remains high; personnel spend a great deal of their own time training with the package. During work hours, a schedule was set up for CAI training. This was necessary so that team members could schedule their training times and so ensure that the entire team received training. An invitation was open for other members of CCSO to train with the package and so allow for technology transfer across the organization.

3.2.3 Video Tapes

Video tapes were found to be effective but were time consuming and did not allow for training flexibility. The SARAH team viewed several different series of tapes. The tapes varied in both quality and effectiveness. Time was allocated for viewing the tapes and attendance varied depending on the workload of the individual team members. The University of Houston video-taped a complete semester course on software engineering and the Ada language. Elements of this course were very beneficial; however quality was poor and it took some time to cover the major language features. A series of tapes named "The World of Ada" provided good background on the Ada language and were useful for manager education. The SARAH team also previewed "Ichbiah, Barnes and Firth on Ada". These tapes provide an in-depth introduction into Ada language syntax and could be used very effectively in an overall training program.

Tapes should be previewed and only the most appropriate tapes should be used for overall team training. To make effective use of video tapes, they should be used in conjunction with other training. Only those tapes which provide good support for particular topic areas should be viewed. A great deal of valuable time can be wasted by subjecting the entire development team to videos that do not necessarily support the overall training approach.

3.2.4 Self-Study

A library of Ada books and materials should be provided for self-study and research. In addition, the development area should have a "quiet area" where people can study without being disturbed. A project library was established soon after the commencement of the SARAH project. Team members set up a data base to control library inputs. The library consists of a collection of Ada articles, catalogs, regulations and books. Today, the library is a very valuable asset to both the SARAH team and CCSO. Team members use the library to remain current with Ada technology advances and to familiarize themselves with various Ada features.

Access to an Ada compiler is necessary if self-study is to be of value. The SARAH team used the self-study method to learn the basic features of the Ada language. There is little point in learning features of the language if they cannot be reinforced by practical exercise problems. A TELESOFT compiler on a Burroughs XE-550 computer was used extensively for this purpose. The self-study method proved to be very effective as a prelude to formal training.

3.2.5 Attendance at Conferences

Attendance at conferences and seminars provided team members with training in the practical application of Ada technology. The SARAH team was active in national and local Ada organizations. These organizations provided a good forum for discussion and allowed members to gain a good insight into some of the problems and pitfalls encountered during software development. The benefits of first-hand experience are not always achievable through formal training alone.

In addition to benefits gained through active participation, conferences often provide free tutorials. The SARAH team has benefited significantly from the tutorials. Topics covered are generally applicable to practical Ada design and implementation. The knowledge gained can help speed software development and enhance software quality.

4. SUMMARY AND RECOMMENDATIONS

4.1. SUMMARY

Software development using the Ada language development environment requires a high degree of training in order to achieve the full benefits designed into the system. Potential Ada developers must gain a practical insight into what is required to successfully develop Ada software. This includes the need to understand and apply the facilities of the Ada language, the various new design methodologies, the Ada Programming support environment tools, and the high startup cost.

Those desiring to become Ada developers should be prepared to take full advantage of the language facilities.These facilities enhance the software engineering concepts for structured programming such as strong data typing, data abstraction, and procedural abstraction. Educators can prudently and reasonably include them in their course structure by properly partitioning the language facilities into coherent training blocks. Other aspects of the software lifecycle such as configuration management, improved documentation techniques, testing, software reuse, and the Ada programming support environment, should be included in a structured training program.

The development methodology selection is based on an analysis of individual needs. Software engineers must be cognizant of related methods and maintain an awareness of current and new efforts in order to take full advantage of improvements in lifecycle application techniques.

The Ada programming support environment provides productivity benefits and will become increasingly significant as more environments become available. The understanding of the environment and its applicability are important for ensuring lifecycle integrity of software.

Ada language training can be acquired from many different places. Training in the Ada language and its associated environment is available through commercial contractors, government, and through self-taught in-house programs. Risks associated with each must be carefully considered. High training costs, the possibility of inadequate or improper training, and procurement problems have been addressed. Care must be taken to ensure the training acquired is worth while and cost effective. In-house training must be undertaken with the greatest of care. The Ada language environment is designed to support the most current and best software programming techniques in use today. These advanced techniques are beyond the capability of previous languages such as FORTRAN and COBOL. For example, in-house training may only teach the Ada syntax and semantics. Programmers will likely recreate FORTRAN or COBOL code in Ada

which may not be as efficient as the original code. In-house training can be effective when using a combination of computer aided instruction, video tapes and self-study to supplement formal qualified training programs.

The Ada language and associated software technology can provide significant benefits in terms of maintainability, software reuse, and programmer productivity. However, managers must be aware that language syntax alone will not provide these benefits. If the Ada language is used without an emphasis on software engineering and without productivity tools, the software produced may be less maintainable and of poorer quality than that developed using older programming languages. Managers must be educated in this new technology if their project teams are to successfully develop Ada software. They must understand that the initial capital investment will be high. Development teams require education in the areas language training, software engineering, development methodologies, and support environments.

4.2. RECOMMENDATIONS

Recommendations are:

- Base Ada training on sound Software Engineering principles.

- Provide up front training for management.

- Research the proposed training organization for instructor experience and approach.

- Ensure that the training investment is sufficient to cover all training needs eg. design, environment, language, management, and software engineering training.

- Provide the development team with CAI packages to help consolidate language training.

- Provide quiet self-study areas. Time should be allocated to allow team members to consolidate their training and to keep current with Ada technology.

- Support the Ada development team. Members will be required to make a significant personal effort if they are to become fully educated in Ada technology.

134

A. BIBLIOGRAPHY

[1]   DRUFFEL L.E., "The Potential Effect of Ada on Software Engineering in the 1980s", North Holland Publishing Company, 1983.

[2]   CARLSON W.E., DRUFFEL L.E., FISHER D.A., WHITTAKER W.A., "Introducing Ada", Proceedings of ACM 80, pp 263-271, 28-30 October 1980.

[3]   "Packages Spawn Ada's Growth", Software and Systems, April 1985, pp 93-100.

[4]   STANLEY R.A., "Whither Ada?", DS&E, March 1985, pp 60-64

[5]   BOOCH G., Software Engineering with Ada, Benjamin/Cummings Publishing, Menlo Park CA, 1983.

[6]   JUDGE J.F., "Ada Progress Satisfies DOD", Defense Electronics, June 1985, pp 77-87.

[7]   "Ada as a Design Language", Ada as a PDL Working Group, IEEE Computer Society, 18 September 1985.

[8]   "SARAH Operational Concept Document", US Air Force, 20 December 1985.

[9]   WAGNER P., "Ada Education and Technology Transfer Activities", ACM Ada Letters, Vol II No 2.

[10]   "Methodman", Ada Joint Program Office, National Technical Information Service (NTIS), accession number AD A123 710.

[12]   BOOCH G., "Object Oriented Development", IEEE Transactions on Software Engineering, Vol. SE-12 No. 2, February 1986.

[13]   CAMERON J.R., "An overview of JSD", IEEE Transactions on Software Engineering, Vol. SE-12 No. 2, February 1986.

[14]   CHERRY G.W., "The PAMELA Designer's Handbook", Thought Tools, Reston Virginia.

[15]   HAILPERN B., "Multiparadigm Languages and Environments", IEEE Software, Vol.3 No.1, January 1986.

[16]   "Requirements for the Programming Environment for the Common High Order Language", Stoneman, Department of Defense, Washington D.C., November 1979.

[17]   "MIL-STD Common Ada Interface Set (CAIS)", National Technical Information Service (NTIS), accession number AD A157-589.

[18]   CONN R., "Overview Of the DoD Ada Software Repository", Dr Dobbs Journal, February 1986.

Ada* for the Manager
A Texas Instruments Perspective

Freeman L. Moore
Texas Instruments Incorporated
Plano, Texas 75086

ABSTRACT

Technical managers are concerned with the management of
people and resources necessary to accomplish a task.
They are not concerned with the implementation details
of a programming language. However, their work is
impacted by the selection of a language, the tools
involved, and the subsequent impact of these and other
items on the budget and schedule. With the
introduction of the Ada programming language comes the
challenge to keep informed about the impact of this
language and implications of its use. Since 1983,
Texas Instruments has been developing its Ada
curriculum to be able to respond to the needs of the
engineering staff within the company. This paper
examines some of the needs of Ada training from the
perspective of a technical manager.

INTRODUCTION

Texas Instruments is a multi-faceted corporation, with one
segment devoted to defense related business, the Defense Systems
and Electronics Group (DSEG). While other segments of the
company are involved in various activities related to the
computer industry, it is the Defense Systems and Electronics
Group that will be most heavily impacted by any requirements
involving the use of the Ada programming language. Even before
language standardization in 1983, Texas Instruments was involved
in some of the Ada language definition activities and continues
to be involved in developing applications utilizing the Ada
language.

----

* Ada is a registered trademark of the U.S. Government, AJPO

Within Texas Instruments, the Computer Systems Training group has been charged with the development and delivery of Ada training products and services since early 1983. Since then, several courses have been developed which address the Ada training needs of the software design engineer implementing solutions using the Ada programming language, and related software support tools. In addition to serving the software engineer, it was recognized that training for program managers would be necessary, but on a scale different from that required by the software engineers. Our current efforts involve working towards integrating the Ada curriculum to address various activities across the entire software life cycle.

THE PROBLEM

"I'm not a programmer" is a phrase commonly encountered when talking with managers about Ada. These are technical people who are involved in the management of programs which are now, or will be, using the Ada programming language and related tools. Some of the early training efforts were directed towards individuals who were involved in software development and implementation. It was noted that a different approach was required when dealing with technical managers. They want to know and understand how this language affects their current method of operation, and what to expect from its use. As a result, the usual programming courses about Ada and software engineering are usually inappropriate for this audience. Most of the programming courses available from commerical training sources appear to be up to five days in length. We have discovered that managers are very stringent about their time, and reluctant to spare more than what is absolutely necessary. We must maximize any training activities within the alloted time available.

Some of the DSEG managers took the introductory programming course and commented on its inappropriateness for their needs. Based upon discussions with these and other managers, the framework for a training program for managers was developed. They agreed that some training to prepare for the introduction of Ada was necessary, but they did not want a programming course, they wanted a "help me understand its impact" course.

TOPICS

It quickly became apparent during our development efforts that certain topics were needed while others could be down-played in a manager's course. Briefly, the topics included:

- **Minimal History:** Awareness of the history does not affect the proper use of the language, nor how to effectively use the language in the future. However, some material detailing why Ada was developed helps to explain some of the rationale of the Department of Defense (DoD) in requiring the use of Ada.

- **Standards:** Even with a minimal background about Ada being presented, it is necessary to understand the various DoD Instructions and Directives and their rationale, that are fundamental to the mandate to the use of Ada.

- **Terminology:** The terminology that is appropriate to the Ada environment must be presented. As a general rule managers must be able to speak the language of their peers, and to be able to effectively communicate using the proper terms. Thus an awareness of the "vocabulary of Ada".

- **Capabilities:** A brief overview of the capabilities and features of the language is needed. Managers don't want to be compared with programmers, but they want to understand what their people are doing. An overview of the language provides the opportunity to reinforce the vocabulary of the language, while giving a high level understanding of the language's features and capabilities. As such, an overview of the features of the language is necessary, but only when presented from the point of view of a non-programmer. In general, identify what is new and different about this language, and in particular, how it differs from the languages that are already in use.

- **Proposals:** Submitting proposals in response to a Request for Proposal (RFP) is a detailed proces˘. and it requires technical expertise in a variety of ʾis ːiplɪnes. When Ada is called for in an RFP, the proposal writer must be informed of the risks and challenges presented by the introduction of this new technology. Proposal writers need help in knowing how to effectively respond to RFP requirements.

- **Design:** With project requirements now including the use of a program design language (PDL), guidelines to understand the

distinction between designing and coding are needed. Technical managers are being exposed to Object Oriented Design and questioning the usefulness of this approach when compared to other approaches, such as functional top-down decomposition. Assistance can be provided in the area of understanding designs, and being able to identify the characteristics good Ada design. Understanding designs should be oriented to the manager and should help to identify what additional training might be needed, how DoD documentation is affected, what tools might be available to support the software development (analysis and design phases).

Computer Systems Training makes available two courses in its Ada curriculum that would benefit the technical manager: (1) a brief seminar introducing the language and its impact and (2) a workshop to help proposal writers respond to Ada requirements.

SEMINAR

To provide an introduction to the impact of Ada, a two hour seminar highlighting the background and capabilities of the language was developed. This has been available to any group that is interested in learning more about Ada, and the resources available within Texas Instruments. This seminar presents some of the reasons on why DoD felt the need to develop a new language, and is augmented by identifying areas where Texas Instruments was involved in the early stages of the language development. A collection of Ada resources at Texas Instruments is also provided.

The history portion provides some reassurance that Texas Instruments has been involved and continues to be active in using this new software technology. By providing a history of Texas Instruments' involvement, attendees also get names and contacts of people / projects that have worked with the Ada language.

Current resources available to projects are identified, including the names of the software support organizations, publications available both internally and externally, and other training possibilities for software engineers. This seminar has been conducted several times, each time further refining the objectives that must to be satisfied, which we continue to incorporate into our evolving curriculum.

RESPONDING TO RFPs

Because of the size and diversity of Texas Instruments, various internal organizations are responsible for developing responses to an RFP. Software managers are typically involved with the software portion, and generally accept all the help they can get on writing the response to an RFP.

RFPs represent a unique challenge to the managers. It is here that they exercise their business skills in addition to their technical skills. When responding to an RFP requiring the use of Ada for implementation or design, the proposal writer must locate resources which can be applied in this situation. Some companies provide management and business courses which address the business portion of proposals, but generally overlook the technical aspects of a proposal, deferring the material to the technical staff.

We determined that specialized training was needed to respond to the technical issues raised in an RFP specifying the use of Ada. These issues might include finding out about similar or related efforts within the company, knowing what compilers and software support environments exist, finding out about the impact of Ada code on size/time problems, understanding how the lifecycle could be impacted, and in general, knowing the resources of the company that are available.

Our response to fulfilling these needs has been the development of a two day workshop addressing the technical issues of proposals, as viewed from an Ada perspective.

Other Training Activities

Internal newsletters provide an additional means of keeping people up-to-date with respect to the current resources available within the company. Newsletters also provide a means of letting other groups know of Ada related activities that are taking place within the same company. However, some material which may be candidates for newsletters may be competition sensitive, resulting in a limited distribution.

STATUS

Texas Instruments responded to the need for training by initially providing a seminar oriented for managers. This training has been supplemented with an internally developed workshop which addresses the needs of proposal writers responding to Ada requirements.

This paper has identified some of the needs of the software and technical managers at Texas Instruments. These people, because of the types of projects they manage, have been or will be affected by the DoD directives requiring the use of the Ada programming language. The internal training division has responded to their needs by making various types of training available internally, along with efforts for providing continuing education.

REFERENCES

Baskette, H., "Life Cycle Analysis of an Ada Project", IEEE Software, January 1987.

Booch, G., Software Engineering with Ada, 2nd ed., Benjamin Cummings, 1986.

Castor, V., "Issues to be Considered in the Evaluation of Technical Proposals from the Ada Language Perspective", AFWAL-TR-85-1100.

Softech, "L201: Ada for Technical Managers", Softech Inc., 1983.

Ada in the MIS World

by

Eugen Vasilescu,Ph.D.
GNV Associates
35 Chestnut St
Malverne NY 11565

Introduction.

The place of Ada in the MIS world is in a state of
flux. There is no doubt that many of Ada features are appealing
for the commercial marketplace. The applications programmer or
the systems analyst will find the concept of package an essential
design tool. This concept allows abstraction of data structures,
separates the specification (roughly the interface) from the body
(the actual implementation), and gives the designer the
possibility to encapsulate objects and the related operations.

Ada's other novel features, like separate compilation, tasks,
exception handling, are clearly significant for systems design
practitioners or managers of information systems.

These features make important contributions to the writing
of portable, reusable software cutting down on the most onerous
data processing activities: maintenance and conversion efforts.

The question, though, is whether Ada is ready for the commercial
world, and, conversely, whether the commercial world is ready
for Ada.

The answer to this question depends lo a large extent on what
Ada has to offer to the many MIS departments using COBOL and/or

various DBMS in conjunction with some fourth-generation languages (4GL).

Ada and Cobol.

To the COBOL practitioners, Ada proponents can always point to the many Ada features already mentioned, and one can add of course all the reasons why Ada as a language is a better breed. The COBOL practitioner can point though to a number of weak spots: Until recently, one could find very few textbooks and other training materials focusing on commercial or business applications of Ada, but this situation is beginning to change (see [Vasi86]). The lack of significant experiences in using Ada in a MIS environment was another weak spot. However, some recent experience with the UNITREP Ada prototype involving the conversion of an existing COBOL application to Ada shows that the lines of code were reduced by a five to one ratio, the resulting application was running faster, the effort in rehosting the Ada application is minimal, and it runs using any of five different compilers. Compare this with the effort in porting a COBOL application from one type of hardware to another, or even from one version of a COBOL compiler to another on the same system.

Finally one can point to the lack of efficient file handling packages in Ada, as opposed to COBOL's established indexed sequential file handling. The Ada I/O packages SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, hardly qualify for building blocks in any demanding commercial application. The glaring ommission of an INDEXED_IO package in the predefined language environment forces the language vendors to supply equivalent implementation dependent packages. This

144

situation has predictable effects on the portability and maintainability claims. This ommission was noted early by the Ada community, and attempts to repair this situation are being made. For instance, in [Kurb86] one can find the specification of an INDEX_SEQUENTIAL_IO package together with a partial implementation of it using B-Trees. It follows closely the specification if DIRECT_IO, with the stated objective to preserve as much upward compatibility as possible (namely, the effect of using the INDEX_SEQUENTIAL_IO with natural keys, should be the same as the use of DIRECT_IO). It does not provide for several indexes, even though the paper mentions this possible extention. Also, In [Wied86] there is the specification of an INDEXED_IO package that handles several indexes and arbitrary kinds of records.

Ada and 4GL.

There is considerable talk, however, about Ada versus 4GLs. While there is no agreed upon definition as to what a 4GL is, one can list among its expected features:

1. It is non-procedural.

2. It is user-friendly.

3. It is used in conjunction with a Database Management System.

4. It aims to integrate several functions (like report preparation, communications etc.)

As there is no standard 4GL, contrasting Ada with any particular 4GL is not a straightforward process.

145

The following observations, however, seem to apply to all 4GL. Software developed with a 4GL is not generally portable from one system to another, while Ada was designed specifically with portability in mind.

A 4GL is best used when a one-time application has to be developed in a short time frame using an existing Database System. The efficiency of a 4GL relates to the speed of developing code. A 4GL, on the other hand, seems less concerned with the running speed of the applications, or the tightness of code generated. A 4GL seems even less concerned with issues that account for the largest cost in the software life cycle: reliability and maintainability. In contrast, Ada design goals emphasized maintainability, reliability, efficiency in using time and space resources.

A 4GL has no explicit requirements for software tools useful in managing the complexity of large systems: data abstraction, information hiding, etc. Some kind of informal and implicit information hiding is achieved by a 4GL because in a non-procedural language the user is relieved of specifying how to perform the desired work. Instead, the user concentrates on defining what work is to be performed. However, one can achieve the same division of labor (and the associated gain in productivity) by designing in Ada appropriate package specifications and bodies for the problems at hand. It is here that one has to look closely and make a reasoned decision. If the situation at hand requires, let us say, rapid prototyping, a 4GL might be chosen. If, on the other hand, one has to deal with a host of similar or related problems, then the cost of developing (or

buying off the shelf) of the necessary Ada packages can be easily
justified.

It seems that a 4GL and Ada do not have to compete, but instead
they might complement each other.

Ada and DBMS.

The field of Database Management Systems is dynamic. The Relational
approach is now coming to dominate the world of commercial offerings.
The relational approach has known advantages due to its theoretical
clarity and flexibility, and recently appeared a number of robust and
efficient implementations. While new DBMS applications will probably
choose a Relational approach, there is a tremendous investment
in past and current implementations that use a Network or
Hierarchical approach. In addition one can see the first
appearances of products using the Entity-Relationship model, and
there is considerable work in progress dealing with topics like
Knowledge-Based Databases, Semantic Data Models, et al.   As far
as Data Manipulation is concerned, the recent ANSI SQL standard
[SQL86] is used by virtually every major Database vendor.

There is a clear recognition though of the need to deal with
both flat structures (standard in the Relational approach) and
hierarchical structures (naturally represented, say, in IMS).
For instance, in [Dada86] one can find the description of a
SQL like language able to integrate flat and hierarchical
structures because it allows relations to occur as attribute
values of tuples in relations. Also, another perceived need
by some DBMS practitioners (as, for instance, in [Bune86]) is

the need for a "class construct", tnat would be persistent (that
is, it survives from one program invocation to another). This
"class construct", in fact includes the concept of type, as
defined in [Ada83].

It is interesting to point out that the Ada/SQL
prototype under development at IDA chooses an approach which will
answer the concerns and needs expressed in [Dada86] and [Bune86]
in the following manner: by allowing as attributes any object of
a particular type, the compiler will take over many chores
associated with expensive maintenance operations; by representing
database columns as components of Ada records, one can choose as
components other Ada records, and one can have this way a handy
mechanism to conveniently represent hierarchical data models.

It is clear then that the use of Ada in the DBMS world can
have far reaching and beneficial applications. In fact,
in [Wied87] the question is raised whether a DBMS should be
written in Ada, and the answer is an emphatic "yes". In [Wied86]
and again in [Wied87], it is advocated the writing of a WIS DBMS
entirely written in Ada. It is further argued that the one time
development cost of an Ada DBMS cannot exceed the costs related
to the modification and tuning of a multitude of commercially
available DBMS. Instead, it is proposed a layered product that
would be carefully modularized and could be taylored to a variety
of configurations. The contemplated modules would be designed
around six layers, each layer encapsulating a given level of
abstraction and functionality. In particular, the first layer
is concerned with Operating Systems services, the second layer
deals with file access methods, the fifth layer includes data

148

manipulation languages like SQL, the sixth and last layer is concerned with applications-oriented packages. In [Wied87] it is further argued that the proposed Ada DBMS would incorporate the Relational, Network, and Hierarchical Models, with other modules reserved for Knowledge-Based Processing, Logic Programming, et al.

On the other hand, one clearly needs to learn more from real implementations of Ada projects having DBMS relevance. Unfortunately, the list of such projects is not long. One may quote here [Spr87], which discusses a large scale embedded system providing DBMS functionality, including a Data Definition Language, a Data Manipulation Language, a Query Language Capability and an Access Control Scheme.

Conclusion.

Ada can be used and is used right now in a MIS environment, as can be seen from the UNITREP prototype and the experiences reported in [Spr87]. Ada could be an immediate and powerful contender in a MIS environment if two critical areas are addressed. One area is the developing of a powerful indexed sequential I/O package. The second area is the developing of an Ada/SQL interface that would take advantage of the strong typing and abstraction building mechanisms of Ada. Once these two areas are covered, one can say that Ada is ready for the commercial world, that it can easily blend and wrap-around existing applications, the long term benefits of using Ada will come into focus, and finally all the elements of a winning strategy

for Ada in the MIS world are in place.

Beyond the immediate horizon, one can see the whole DBMS world coming to appreciate and use features best embodied in Ada. In fact, the arguments for building an Ada DBMS point to a product ideally suited to accomodate present and future Data Modeling approaches.

References

[Ada83]
Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A,United States Department of Defense,
Washington DC, January 1983.

[Bune86]
Buneman,P. and Atkinson,M.," Inheritance and Persistence
in Database Programming Languages ",Proc. of SIGMOD'86,
SIGMOD, Volume 15 Number 2, June 1986

[Dada86]
Dadam,P.,et al.,"A DBMS Prototype to Support Extended
Non First Normal Form Relations: An Integrated View on Flat
Tables and Hierarchies", Proc. of SIGMOD'86, SIGMOD Volume 15
Number 2, June 1986

[Kurb86]
Kurbel,K. and Pietsch,W,"A Portable Ada Implementation of Indexed
Sequential Input-Output", Ada Letters, Volume VI, No. 2,3, 1986.

[SQL86]
ANSI Database Language SQL, American National Standards
Institute, Inc., NY, NY, 1986.

[Spr87]
Springer,M.L. and Lawson,R.L." The AFATDS Data Management
Software System", Proc.of Joint Ada Conference Fifth National
Conference on Ada Technology and Washington Ada Symposium,
March 1987.

[Vasi86]
Vasilescu,E.,"Ada Programming with Applications",Allyn and Bacon,
Newton, Mass., 1986.

[Wied86]
Wiederhold,G. et al.,"Software Requirements for WIS Database
Management System Prototypes".

[Wied87]
Wiederhold,G,"DBMS System Requirements and Ada",STARS Briefing,
Washington, March 1987 .

This page left blank intentionally

Teaching COBOL Programmers to be Ada* Software Engineers

by
Major Charles B. Engle, Jr.
&
Major Colen K. Willis

INTRODUCTION

Although the title of this paper may seem "slightly" pejorative, it is not our intention to denigrate COBOL (Common Business-Oriented Language). Like an old dog, it has served us well and should be remembered for what it was and what it did for us, not what it doesn't do or can't do for us. Therefore, let us concentrate on a problem which will be a major one for DoD and perhaps, to a lesser extent, to those outside DoD. The problem is how do we convert programmers with long careers in COBOL to the mindset of software engineering with Ada. The United States Military Academy at West Point has been involved in Ada education and training since 1979. Besides offering an advanced level elective in Ada to cadets majoring in computer science, the Military Academy also offers an annual Ada *Summer* Workshop for DoD and military programmers. The Ada Summer Workshop is designed to expose participants, primarily trained and employed as COBOL programmers, to Ada through an intensive two week course. The syllabus used in our Ada Summer Workshop is found in Appendix A. The focus of this paper will be the lessons we learned in two different iterations of the workshop. It is our intention to help those that follow us avoid the pitfalls that we experienced and capitalize on the successes that we enjoyed.

THE SETTING

The number of participants in this course was limited to twenty DoD civilian and military programmers. Their programming experience was rather limited. However, the range of experience was rather large. It extended from a participant that had a Bachelor of Science degree in Computer Science to a junior military programmer who was newly graduated from the fourteen week Army programmer's school. The observations made in this paper are based on the performance of the "center of mass," not on any particular student. The main facility used was the Digital Equipment

* Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

Corporation VAX 11/780 operating under the VMS operating system. The Ada compiler was the DEC VAX Ada compiler, version 1.0. In addition, some selected students were assigned to IBM PC/AT microcomputers operating under DOS 3.1. These students used the Alsys Ada compiler, version 1.0. No significant difference in learning was detected for students using either of these configurations. Each student was provided with a copy of the textbook, Programming in Ada, by J.G.P. Barnes, and with a copy of the Ada Language Reference Manual. Instruction consisted of both classroom discussion as well as lab periods and exercises. The students were required to complete six out-of-class programming assignments.

LESSONS LEARNED

Lesson #1 - Mindset

While we had anticipated open minds and receptive attitudes, we found that the COBOL programmers arrived with preconceived notions about Ada (generally bad) and a strong, almost religious devotion to COBOL as THE programming language. Their approach to programming often included a great deal of dependence on the operating system and a heavy integration of the COBOL language and facilities in the operating system command language. From a systems point of view, their approach was that of a programmer and user, not an engineer and synthesizer. We found it to be one of our fundamental tasks to transition this mindset to a regard for the costs of a program throughout its life cycle. There appears to be no clear way to prevent this mindset, but we found that it may be countered by always discussing Ada concepts in terms of their rationale so as to provide a clear understanding of the reasons behind the concepts. This appeared to work because we noticed a change in the mindset at approximately the half way point in the course. Students were more receptive to new ideas and seemed to appreciate the power of the language and the software engineering concepts behind the language. By the end of the course, there was only one remaining "Doubting Thomas!"

Lesson #2 - Fundamental Computer Science Concepts

It became obvious early in the course that the majority of the students, especially the enlisted military programmers trained in a fourteen week course, had learned a "comfortable" subset of COBOL through rote memorization. These students displayed a lack of knowledge about even the most basic of computer science concepts. This is attributable to the background of these particular participants, but is reflective of DoD COBOL programmers in general. This extends to their knowledge and use of Job control languages, which demonstrated that they knew how to set up Job Control Language (JCL) processes, but did not understand why they were doing so, or even what they were doing in particular. By the end of the third day, we found that we had to divert from the planned syllabus to provide the students with these rudimentary concepts of computing. One might consider providing this instruction "up front" in future courses to save time and prevent disruption of continuity during the course caused by this sidelight instruction. This initial instruction might include such simple concepts as data structures, algorithm analysis, and simple numerics. These are basic issues which are integral to the design and use of Ada as a language and are fundamental to the software engineering context in which we design and use Ada programs.

Lesson # 3 - Scope and Visibility

In common practice, a COBOL program is a large monolithic entity which has a very flat program structure. All variables are global and must be uniquely named. Therefore, when COBOL programmers are exposed to a language in which the

meaning of an identifier depends upon its location in a program, they are initially confused. The idea of potential visibility (scope), and whether or not reference may be made to a variable (visibility), is foreign to these students and needs careful explanation and numerous examples. In our two week course, this was the first area in which there was not a directly analogous concept in COBOL. In order to overcome this COBOL mindset, we tried to draw the analogy between scoping in Ada and record levels in COBOL. Although these are similar ideas, the analogy is not clean. However, it does provide the student with a basis in something that is familiar to begin to understand these new concepts.

Lesson # 4 - Data Types

The overall philosophy underlying the concept of data types was a new one to these students, yet they had relatively little difficulty in comprehending this concept. The idea that a type was a set of values that an object may take and a set of operations upon those values was almost intuitive for the pre-defined types. The concept of strong typing and name equivalence for types was similarly grasped without great pain on the part of the student. The predefined types were explained by referring to the picture clause in COBOL. The student understood that when they described an identifier with a picture clause they were specifying the "kinds" of values that could be stored in that identifier. The various "kinds" of data were referred to as types. The predefined types all have equivalent types in COBOL, so the student readily accepted and understood these Ada terms. The user defined data types caused a great deal more difficulty. The concept of enumerated types, and the ability to manipulate user defined identifiers as "values" of objects declared to be on an enumerated type proved to be a rather perplexing idea. There is no analogous concept in COBOL. Similarly, when we introduced the structured data types, the students seemed intimidated. Although the idea of one-dimensional arrays is exactly the same in COBOL as in Ada, most of our students had not programmed with COBOL's tables and thus, had considerable difficulty with this concept. Further, in practice, COBOL programmers rarely use (need?) multi-dimensioned arrays. Therefore, when this structured data type was introduced, we were required to put our planned syllabus on "hold" and explain this data structure from the beginning, i.e., we had to justify the need for multi-dimensions and relate them to matrices. The Record is another difficult data type for the student to comprehend. The COBOL programmer treats records as I/O facilities, whereas in Ada they are treated as data structures. We found that we were required to be very precise in our handling of this data structure to minimize confusion. Needless to say, discriminated records and variant records were introduced, but could not be adequately explained in the short time we had for this course. The lesson to be learned here is that structured data types and, indeed, the whole concept of data types, are new concepts for COBOL programmers. In hindsight, it would have been more appropriate and effective to include the ideas underlying these data structures in the computer science concepts instruction which, as already mentioned, should have been required prior to the introduction of Ada syntax, semantics, and philosophy.

Lesson # 5 - Building Programs

We found that our students learned the "art of engineering programs" through the use of simple examples. Once we felt that the students understood the basic ingredients of a "module", we immediately began to form programs using modularity. In order to accomplish this the programmer needed to understand the concept of modularity through packages. Students were encouraged to "stepwise refine" their program designs, even in small programs, into functional modules. We found that the "center of mass" was not experienced in "independent compilation" of COBOL units, however, surprisingly, most of the students had little difficulty in understanding Ada's separate compilation and library management. The lesson we learned is that these concepts build upon one another and should be introduced in a logical order culminating in the student's understanding of reusability, modularity, and library management. This topic is the one which seemed to start to "win their hearts and minds" over to Ada.

Lesson # 6 - Input/Output

The students had little difficulty with the differences in input/output facilities between COBOL and Ada. The concept of overloading was not found to be a difficulty, when presented from the point of view that a put with a data type was like a move in COBOL with any data type. The idea of using the same name for a series of different, but similar operations, and having the compiler determine which of the various meanings of the operation was desired from the types of the arguments, seemed to cause little consternation. In fact, the student's major complaint about I/O was the requirement to instantiate a generic package for any form of numeric I/O (using 'Image is cheating!). Their next biggest concern was the lack of built-in facilities for Indexed Sequential I/O. The lesson learned here is that I/O can be presented in a rather straightforward manner without difficulty.

Lesson # 7 - Abstraction

We separated abstraction into data abstraction, using packages and private types, and procedural abstraction, using procedures and functions. The concept behind the private type, in both of its forms, is new to the student, but is not difficult to explain if suitable examples are used. Thus, data abstraction was not found to be a problem area. Procedural abstraction was a more complex idea for the student to grasp. Procedures were contrasted to paragraphs and the analogy helped to explain their purpose. However, the idea of parameters to procedures was new and it proved to be a difficult concept for the students to grasp. Several of the classic approaches to introducing parameters (suggested by several authors from both the Ada and Pascal worlds) were used. Nonetheless, the students struggled with this concept. Functions were likewise very difficult for most of the students to understand. There is nothing in COBOL analogous to function subprograms, so the idea of a "subprogram within an expression" was doubly difficult. The lesson we learned here is that an effective method of teaching procedural abstraction is to contrast procedures to paragraphs and to attempt to explain parameters as arguments to the Call Using statement in COBOL.

The usual difficulties encountered in teaching the subject of parameters at the introductory level should be expected.

Lesson # 8 - Dynamic Data and Recursion

As might be expected, the subject of recursion caused the students confusion because their is no parallel in their programming experience since COBOL does not allow recursion. Dynamic data such as access types was also beyond the level of most student's understanding. Those that were able to get their lab program on access types operating, were using their notes by rote and getting inordinate amounts of additional instruction from the instructors. The lesson we learned is that these concepts are perhaps too advanced for this type of student, without formal background, to grasp in a two week course.

Lesson # 9 - Ada's Advanced Features

Surprisingly, the concept of generics did not prove to be one that caused the students any difficulty. The ideas are new and, when adequately explained, are readily used by the students. The fact that all non-textual I/O must be instantiated caused the student to become familiar with the instantiation portion of generics throughout the course. Thus, when the students were shown how to design and write generic units, they had an appreciation of how to use them, which made it easier for them to visualize and use the concept. Exceptions were also new to the student, but also did not pose any problem for the students. Perhaps because of their analogy to the COBOL On End statement for I/O, the idea was readily accepted by the students and easily grasped. Tasks posed a problem because of the students' unfamiliarity with the concept of concurrent processes. Also, this concept was presented late in the course and was not given too much in detail. Students did not seem to be able to understand the basic problems, so Ada's elegant solutions to these problems were not grasped at all. The lesson to be learned is that exceptions can be made analogous to the COBOL On End statement. This facilitates the student's understanding of the new concept. The other two area, generics and tasks, are completely new and need to have extra time devoted too them. Generics will be more easily grasped and used by the students then will tasking.

Lesson # 10 - Advance Assignment

The lesson that we learned which proved to very valuable was to send out the final Ada programming assignment in advance. We required each student to program their solution to the assignment on their home station's computer, using whatever language they felt most comfortable in using. We then had the students hand this assignment in prior to the start of the instruction and planned our programming labs throughout the course so that they built up to the final programming lab. After the students handed in the last lab, we critiqued it and handed back their advance assignment to compare and contrast the two versions Students felt that they gained most from a comparison of "before and after."

SUMMARY

This was a very unscientific listing of the fun and folly that we had in our two iterations of teaching Ada philosophy and programming to COBOL programmers. It is not intended to be an exhaustive comparison of the languages, but rather a listing of the way we taught the course and some of the lessons that we learned. We sincerely hope that instructors in similar courses might find some value in the suggestions that we have made.

APPENDIX A

USMA 1986

ADA SUMMER WORKSHOP

SCHEDULE

-------------------------- 16 JUNE - MONDAY --------------------------------

TIME	GROUP	LOCATION		TOPIC	INSTR	ASSIGNMENT
0800-0830	All	6004	1.1	Introductory Remarks	K	N/A
0830-0900	All	All		Tour of Facilities	E/W/G	N/A
0910-1000	All	4086	1.2	CS/Data Structure Review Ada Language Overview "A Software Crisis"	E	B: Ch 1
1020-1130	1	4086	1.3	Introduction to Ada Introduction to LRM "An Ada Program"	W	B: Ch 1&2 DoD: Ch 1
	2	5038A	1.4	Introduction to USMA Computer Systems LAB 1	G	N/A
1130-1300	ALL	O-Club		Lunch		
1315-1420	1	5038A	1.4	Introduction to USMA Computer Systems LAB 1	G	N/A
	2	4086	1.3	Introduction to Ada Introduction to LRM "An Ada Program"	W	B: Ch 1&2 DoD: Ch 1
1430-1520	1	4086	1.5	Lexical Style	E	B: Ch 3,4 B: App 4 DoD:Ch 2-4
	2	5038A		LAB 2 (Lexical Style)	G	
1530-1630	1	5038A		LAB 2 (Lexical Style)	G	
	2	4086	1.5	Lexical Style	E	B: Ch 3,4 B: App 4 DoD:Ch 2-4
1630-1700		5038A 5038B		Independent Study		

```
-------------------------- 17 JUNE - TUESDAY --------------------------
TIME          GROUP    LOCATION           TOPIC              INSTR    ASSIGNMENT
--------------------------------------------------------------------------------

0800-0830     All      4086           Admin - Facility Tour    W      N/A

0830-0920     All      4086     2.1   Simple File I/0          W      B:15.1-15.
                                                                      DoD: 14.3

0930-1030     1        4086     2.2   Control Structures       W      B: Ch 5
                                                                      DoD: 5.3-
                                                                           5.9
              2        5038A          LAB 3 (File I/O)          G

1040-1140     1        5038A          LAB 3 (File I/O)          G

              2        4086     2.2   Control Structures       W      B: Ch 5
                                                                      DoD: 5.3-
                                                                           5.9

1315-1400     All      4086     2.3   Composite Types  1       E      B: Ch 6
                                                                      DoD: 3.6-
                                                                           3.7.4
                                                                           5.2.1

1410-1500     1        4086     2.4   Composite Types  2       E      (See 2.3)

1510-1600     2        4086           LAB 4 (Complex Types/    G
                                      Control Structures)

1510-1600     1        5038A          LAB 4 (Complex Types/    G
                                      Control Structures)

              2        5038A    2.4   Composite Types  2       E      (See 2.3)

1600-1615     All      4086           Admin                    E

1615-1645     All      4086           Quiz 1                   E

1645-1700     All      5038A          Independent Study
                       5038B
```

APP A

161

```
----------------------------- 18 JUNE - WEDNESDAY -----------------------------
TIME          GROUP    LOCATION          TOPIC              INSTR        ASSIGNMEN
-------------------------------------------------------------------------------

0800-0900     All      4086        3.1  Subprograms  1       W           B: Ch 7
                                                                         DoD: Ch 6

0910-1000     1        4086        3.2  Subprograms  2       E           B: Ch 7
                                                                         DoD: Ch 6

              2        5038A            TEC Center Tour       W

1000-1100     1        5038A            TEC Center Tour       W

              2        4086             Subprograms  2        E          B: Ch 7
                                                                         DoD: Ch 6

1100-1145     All      5038A            Independent Study
                       5038B

1315-1405     All      4086        3.3  Packages  1           W          B: Ch 8
                                                                         DoD: Ch 7

1415-1505     1        4086        3.4  Packages  2           W          B: Ch 8
                                                                         DoD: Ch 7

              2        5038A            LAB 5 (Packages)       G

1515-1605     1        5038A            LAB 5 (Packages)       G

              2        4086             Packages  2            W         B: Ch 8
                                                                         DoD: Ch 7

1605-1620     All      4086             Quiz 2                 W

1715-2030     All      South            Boat - Trip                     N/A
                       Dock
```

```
----------------------------------- 19 JUNE - THURSDAY ------------------------

TIME          GROUP     LOCATION            TOPIC                  INSTR     ASSIGNMEN

------------------------------------------------------------------------------------

0800-0815     ALL       4086                Admin                  E         N/A

0815-0905               4086            4.1 Library Units          W         B: Ch 8
                                                                             DoD: Ch 1

0915-1005     1         4086            4.2 Exception Handling     E         B: Ch 10
                                                                             DoD: Ch 1

              2         5038A               LAB 6-1                G
                                            (Comprehensive)

1015-1105     1         5038A               LAB 6-1                G
                                            (Comprehensive)

              2         4086            4.2 Exception Handling     E         B: Ch 10
                                                                             DoD: Ch 1

1105-1145     All       5038A               Independent Study

1351-1415     All       4086            4.3 Private Types          W         B: Ch 9
                                                                             DoD: 7.4

1430-1500     All       4086                Quiz 3                 E

1510-1610     All       5038A               LAB 6-2                G
                        5038B               (Comprehensive)

1620-1700     All       5038A               Independent Study
                        5038B               LAB or TEC Room
```

APP A

```
--------------------------------------- 20 JUNE - FRIDAY ----------------------------
```

TIME	GROUP	LOCATION	TOPIC	INSTR	ASSIGNMENT
0800-0900	All	4086	5.1 Advanced Types 1	E	B: Ch 11.1 11.2 DoD: Ch 3.
0910-1000	1	4086	5.2 Advanced Types 2	W	B: Ch 11.3 11.6 DoD:3.2,3. 3.6.3
	2	5038A	LAB 7 (Discriminated Records)	G	
1010-1100	1	5038A	LAB 7 (Discriminated Records)	G	
	2	4086	Advanced Types 2	W	B: Ch 11.3 11.6 DoD:3.2,3. 3.6.3
1100-1145	All	5038A 5038B	Independent Study		
1315-1405	All	4086	5.3 Advanced Types 3	E	B: Ch 11 DoD: 3.8
1415-1445	All	4086	Quiz 4	E	
1545-1700	All	5038A 5038B	LAB 6-3 (Comprehensive)	G	

APP A

164

```
-------------------------------- 23 JUNE - MONDAY --------------------------

TIME          GROUP    LOCATION           TOPIC                  INSTR   ASSIGNMENT
------------------------------------------------------------------------------------

0800-0850     All      4086       6.1  ARRAYS AND RECORDS        E/W     B: Ch 6
                                                                         DoD: 3.6-
                                                                              3.7

0900-0950     1        5038B      TEC ROOM                       W
              2        5038A      LAB 6 (Comprehensive)          G

1000-1050     1        5038A      LAB 6 (Comprehensive)          G
              2        5038B      TEC ROOM                       W

1100-1130     All                 Independent Study

1315-1405     All      4086       6.2  Access types             E       B: Ch 11.3
                                                                              11-5
                                                                         DoD: 3.8

1415-1430     All      4086       Issue/Explain LAB 8           E

1430-1700     All      5038A      Independent Study
```

APP A

165

```
-------------------------------- 24 JUNE - TUESDAY --------------------------

TIME          GROUP    LOCATION          TOPIC                 INSTR    ASSIGNMENT
----------------------------------------------------------------------------------

0800-0850     All      4086      7.1  Numeric Types            E       B: Ch 12
                                                                       DoD: 3.5.4
                                                                             3.5.1

0900-0950     1        4086      7.2  Files                    W       DoD: Ch 14
                                           Advanced Format
                                           Techniques

              2        5038A          Lab 8 (Access Types)  G

1000-1145     All      5038A          Independent Study

1315-1405     1        5038A          LAB 8 (Access Types)  G

              2        4086      7.2  Files                    W       DoD: Ch 14
                                           Advanced Format
                                           Techniques

1405-1700     All      5038A          Independent Study
```

APP A

166

```
------------------------------------- 25 JUNE - WEDNESDAY -------------------------------
 TIME          GROUP      LOCATION            TOPIC             INSTR      ASSIGNMEN
-----------------------------------------------------------------------------------------

0800-0850      All        4086         8.1  GENERICS  1          E        B: Ch 13
                                                                          DoD: 12.1
                                                                               12.3

0900-1000      1          4086         8.2  GENERICS  2          E        B: Ch 13
               2          5038A             LAB 9 (Generics)     G        DoD: 12.1
                                                                               12.3

1000-1130      All        5038A             Individual Study
                          5038B

1315-1405      1          5038A             LAB 9 (Generics)     G

               2          4086         8.2  GENERICS  2          E        B:Ch 13
                                                                          DoD: 12.1-
                                                                               12.3

1405-1430      All        4086              Quiz 5               E

1430-1700      All        5038              Independent Study
```

```
-------------------------- 26 JUNE - THURSDAY --------------------------

TIME          GROUP     LOCATION            TOPIC                INSTR     ASSIGNMENT

-------------------------------------------------------------------------------

0800-0850     All       4086           9.1  TASKING 1            W         B: Ch 14
                                                                           DoD: 9.1 -
                                                                                9.12

0900-1000     1         4086           9.2  TASKING 2            W         B: Ch 14
                                                                           DoD: 9.1 -
                                                                                9.12

              2         5038B               LAB 10 (Tasking)     G

1000-1130     All       5038B               Individual Study

1315-1405     1         5038B               LAB 10 (Tasking)     G

              2         4086           9.2  TASKING 2            W         B: Ch 14
                                                                           DoD: 9.1 -
                                                                                9.12

1405-1430     All       4086                Quiz 6               W

1430-1700     All       5038                Independent Study

1800-????     All       TBD                 Farewell Dinner      E/W/G
```

```
------------------------------ 27 JUNE - FRIDAY ------------------------------
TIME        GROUP    LOCATION           TOPIC              INSTR    ASSIGNMENT
------------------------------------------------------------------------------

0900-0950   All      4086      10.1  Language Discussion   E W G    TBA

1000-1030   All      4086      10.2  REVIEW                E W G    TBA

1030-1050   All      4086      10.3  Course Critique       E W G    N/A

1100        All      6004            Graduation              K

1200        All                WORKSHOP ENDS
```

INSTRUCTORS

MAJ Chuck Engle   (E)
MAJ Colen Willis   (W)
LCDR Al Gary   (G)

REFERENCES

Barnes,J.G.P; "Programming in Ada"; 2nd Edition;   (B)
   (Addison-Wesley Publishing Co, London,1984)

United States Department of Defense; ANSI/MIL-STD-1815A-1983;   (DoD)
   "Reference Manual for the Ada Programming Language"

This page left blank intentionally

TEACHING SOFTWARE ENGINEERING IN A FIRST ADA COURSE

Robert C. Mers

North Carolina A. & T. State University

Abstract

The author has been teaching an Ada language course at North Carolina A. &
T. State University for five consecutive semesters beginning in spring 1985.
From the very beginning, this course has assumed a complete knowledge of Pascal
as a prerequisite, including knowledge of pointers and variant records. This
course gives a complete overview of the Ada language, covering elementary topics
in depth and topics such as tasking, generics, and user defined real types
somewhat hurriedly. Within the past year this course has been upgraded from CS
290, Programming in Ada , to CS 490, Software Engineering Using Ada , as part
of the upgrade of the Computer and Information Science program to meet
accreditation standards of the CSAB (Computer Sciences Accreditation Board) of
the ACM and the IEEE. An additional prerequisite of data structures is now
required. The purpose of his paper is to describe the development of this
course and the use of program assignments to (1) teach software engineering
concepts such as program design, modularity, data abstraction, encapsulation of
related data into packages, separate compilation, data integrity, information
hiding, generics and reusability, and exception handling, and (2) develop the
Ada language constructs themselves.

1. Introduction and Context

This paper could be considered a sequel to the author's paper [5] titled
Experiences of Pascal Trained Students in an Introductory Ada Course . Whereas
the emphasis there was on student experiences, the emphasis here is on teaching
methodology and upgrade of the course. An overview of the course is given,
focusing on the progression of topics. The utilization of programming
assignments to teach students software engineering concepts and methologies is
then described. The culmination of the course is a team project that challenges
the students to use many of the software engineering methodologies in a single
application, and some examples of projects are given. Resources are also
described. The author's evaluation of his efforts to upgrade follows,
mentioning difficulties and limitations as well as successes so far. Finally
conclusions and recommendations are mentioned, and an open invitation is given
for colleagues to offer suggestions to further enhance such a course.

The course described is in the context of an environment where Pascal is the
main language, and proliferation of languages is discouraged. The Department of
Mathematics and Computer Science gave the author permission to teach Software
Engineering using Ada, not Ada as a language. Thus the issue of Ada being the

171

primary language is not addressed. Limitations such as a shared VAX 11-780 and
an error prone New York University Ada Ed Compiler are dealt with, for these
have prevented the instructor from implementing this course to its maximum
potential.

2. Overview of Course Content

The first part of the course is an overview of software engineering
principles designed into the Ada Language prior to giving of nitty gritty
details of Ada. Rationale of the Ada language is given along with a brief
description of the major goals and features of Ada (see Barnes [2]). These
include data abstraction, separation of specification and implementation,
packages and separate compilation, data integrity, generics, and parallel
processing. A history of the Ada language is presented from the beginning to
current developments. Then a fairly detailed treatment of packages,
subprograms, generics, and TEXT_IO is given so that students can understand
simple Ada code in its larger context and the use of the TEXT_IO package.
Illustrations of Ada code are given, for these students have the programming
background to read such code without much difficulty.

Since this is a first Ada course, the rudiments of the language are covered
next, including such topics as control structures and discrete types. Because
of their programming background, the students have little difficulty with
declarations, operations, and enumerated types. Subtypes, derived types,
attributes, and type conversion come next. The emphasis on software design
continues as detailed features of subprograms, such as named and positional
notation, recursion, default parameters, and overloading of subprogram names and
operations, are covered.

The course now makes the transition from simple to compound data types.
Declare blocks are introduced here because of their application to creation of
array objects at run time. Both constrained and unconstrained array types are
introduced, as are array attributes, assignment using aggregates, slices, array
operations, and strings. Records without discriminants are covered in detail.

Emphasis is now shifted to the use of external packages and separate
compilation. Compilation units and order of compilation are covered. Data
integrity is made concrete with the introduction of private and limited private
types. Scope and visibility are covered in depth, as is dot notation, the use
clause, and the features of renaming.

At this point the students appear to have enough confidence in themselves to
handle higher level software engineering concepts such as generics and exception
handling. Predefined and IO exceptions, exception handling, and propagation are
covered in some detail, and students are encouraged to use exception handlers in
programs from this point on. Generics are covered in depth, the students being
made very aware of reusability of code through examples such as generic sort
subprograms and generic stack packages.

The advanced typing features covered next, such as discriminated records and
access types, build on the previous treatment of types. Students are then

172

Introduced to user defined floating point and fixed point types, including their hardware implementations. The author has tried introducing generics and exception handling both before and after these advanced typing features. He feels that covering generics and exceptions first is more in tune with the software engineering tone of the course.

Tasking gets covered last due to its uniqueness and unfamiliarity. The concepts of parallel processing, rendezvous, entry and accept statements, select, delay, and terminate statements do get covered. However this topic and user defined real types do get short changed. The author feels that in-depth tasking may be more appropriately covered in an advanced systems course and that real types can be explored further in a numerical analysis setting.

3. Individual Program Assignments

The author has found that the hands-on aspects of a programming course teach a student far more effectively than lecture or discussion, for the student is involved in the *doing* dynamic. And as we all know, programs work perfectly in our minds but will not compile or run when put in the machine. Programming assignments are intentionally created to gradually develop the student's ability to use the software engineering features of the Ada language. A major objective of the individual programming assignments is to prepare the students for a team project using almost all of the advanced Ada features and software engineering concepts.

The first program assignment develops the student's ability to do TEXT_IO, work with various scalar types, and do separate compilation from libraries. Normally a package is provided for a student, and he or she examines it and writes a driver program to implement its resources. In Fall 1986 the instructor provided a package with Sine and Cosine and had the students compute all trigonmetric functions of angles from 0 to 360 degrees in increments of 10 degrees and handle the cases where the trig functions are undefined. Later on the students could enhance the program by writing exception handlers for undefined functions and using generics on user defined floating point types. In Spring 1987 the students had to apply Exponential and Logarithmic functions provided them to solve problems involving compound interest. They used Enumeration types to imput the frequency at which interest was compounded.

If the pace of the course goes rapidly enough, the students may be ready to handle array types, including unconstrained arrays, and simple records in a second programming assignment. Otherwise, another assignment can be given on scalar types, emphasizing attributes of discrete types, followed by an assignment involving compound data types. By the second or third assignment the students are required to design their own packages containing low level routines and reflecting the levels of abstraction in the program. In Fall 1986 the instructor had the students develop the operations of addition, scalar multiplication, and dot and vector products on unconstrained arrays and provide error handlers when the operations were undefined. This assignment provided them the chance to use overloaded operators. In Spring 1987 the students had to process an unconstrained array of records, including updating of information and sorting on various fields of the record.

173

The instructor has found that students have not developed the sophistication to handle private types, rigorous exception handling, or generics until late in the semester and after at least the above mentioned assignments are completed. Although the emphasis is on software engineering concepts, this is a **first** Ada course, and it takes time to integrate all the major concepts of the language into a single programming assignment. If another individual assignment is given, it may focus on access types or discriminated records and emphasize exception handling and data integrity.

4. Team Projects

Approximately one month before the end of the semester, the students form groups of two or three for their major team project. Teams may choose their own topics subject to instructor approval or select from a list the instructor provides. All the projects require extensive use of software engineering principles such as data abstraction and information hiding, data integrity, reusability, and robustness. Careful design into external packages and use of separate compilation units is required, as is exception handling. Use of generics and private types is encouraged if feasible for the project chosen. At the time of completion, the team must give an oral presentation to the instructor, enabling him to determine the level of understanding and contribution of individual team members. The projects generally involve 200 to 400 lines of code. However, the level of expertise with software engineering principles is a more important criterion in the project than length or complexity.

So.e of the projects done by students and evaluation of them are now described.

(1). A generic transecendental function package which works on any floating point type. The students wrote correct algorithms for square root, natural logarithm, and exponential functions, raising and handling exceptions when the functions are undefined.

(2). A queue package using access types and consisting of procedures to locate, insert, remove from front, and remove from any position in the queue. Exception handlers were provided for items not found in the queue or removing from an empty queue. The program was innovative and worked well; however a compiler bug prevented it from being executed as a generic package.

(3). A generic numerical analysis package that included integration by the trapezoidal rule. Functions used included transcendental functions, and various exception handlers were provided. Execution was successful in the non generic case; however a compiler bug prevented execution using this generic package. which had a subprogram as a formal parameter.

(4). A binary search tree package using access types which included a recursive infix traversal and iterative insert and delete routines that

174

preserved the sorted structure of the tree. The delete procedure was especially innovative.

(5). Processing of a circular doubly linked list, again using access types. Nice exception handlers were provided for deleting from empty list, deleting item not in list, and avoiding duplicate insertion.

(6). A matrix operations package involving standard operations such as addition, multiplication, determinant, and inverse. This package allowed the user to choose operands from an array of matrices, and was quite detailed. However, it lacked robustness in failing to handle undefined operations and did not use the power of recursion for the determinant.

Among other projects attempted, with less success, were a generic quicksort procedure, a singly linked list package, a postfix expression evaluation program, nested linked lists, and a polar coordinate complex number package. These projects were not completed because these students did not exhibit a high enough level of understanding to get rid of compilation and other errors. Differing levels of success is not surprising due to varying levels of ability and commitment among the students.

4. Resources

It is necessary to mention the resources and environment, especially the textbook and compiler, for the quality of these has greatly affected the success of teaching software engineering concepts in this course.

The compiler used was the New York University Ada Ed version 1.7 for the Digital VAX 11-780, which was much faster than versions 1.1 and 1.5 previously used by the instructor, yet full of errors. Some of the errors discovered were (1) failure to handle End_of_File on input done from the VMS editor, (2) failure to sort strings correctly, and (3) run time aborts when formal generic subprogram parameters were attempted. Such errors caused frustration and hindered giving of an adequate number or quality of programming assignments. Breakdowns of the VAX 11-780 also prevented the course from achieving its desired objectives, for the students did not get adequate hands-on time needed.

Due to a proposal coauthored by Dr. Mers and Mrs. Gloria Phoenix of North Carolina A. & T. State University, an almost dedicated VAX 11-785 system and a Digital VAX Ada compiler will be available for future offerings of this course. These improvements will greatly enhance the quality of this and other Ada offerings.

The primary text in the course is Young [7]. Although Booch [3] may have the most comprehensive treatment of software engineering from an Ada perspective, the instructor feels that the Ada features themselves are buried too deeply in the text and that the approach is too advanced for a first Ada course, even for senior level students. Another excellent Ada software engineering text is Bray [4], which uses a bottom up approach. The instructor feels that although this book is very readable and in the spirit of software

175

engineering, it covers many topics in a sketchy manner and tries to do too much too soon (e.g. all types in one short chapter). Although Young's text does not explicitly mention "software enginering", the author feels that it is the best Ada text available from the standpoint of completeness, style, readability, consistency in clarity of presentation, and applications.

Use of the Reference Manual for the Ada Programming Language [1] is encouraged, and students are given the opportunity to purchase copies. Much of the course material is from the instructor's lecture notes, much of which are compiled from SofTech's notes for the U.S. Army Summer Faculty Research Program of 1983 [6] and the Advanced Ada Workshop sponsored by ASEET in January 1987.

5. Evaluation, Conclusions, and Recommendations

The instructor of the course feels that, with a high level production compiler such as Digital VAX Ada and a dedicated machine that does not have frequent breakdowns, that students experienced in programming who have had no prior Ada experience can develop both conceptual understanding of software engineering principles and ability to apply them in hands-on programming. The compiler and hardware limitations mentioned in section 4 above have prevented current implementations of the course from fully achieving its objective of making students familiar with the major software engineering principles of the Ada programming language because the students have not developed sufficient hands-on proficiency. However, the lecture portion of the course has provided the students in-depth exposure to the major software engineering principles of the language except for tasking and real types, and students have demonstrated understanding of these through examination performance.

The author makes the following conclusions and recommendations based on his experiences teaching software engineering is a first Ada course.

(1). The syntax of the Ada language cannot be slighted when teaching software engineering principles. A primary or secondary text must be used which gives a complete, elementary treatment of the language.

(2). Use of packages and separate compilation in programming design should be taught from the very beginning and be the framework in which all programming assignments are done. Data abstraction and information hiding should be emphasized also. Robustness should be encouraged by doing exception handling early.

(3). Private types should be introduced at the time applications to data strutures (e.g. stacks, queues) are introduced so that students can experience data integrity. Perhaps students should write parallel code using private and non private types and then try to see the effects of compromising the data structures used.

(4). A spiral approach to the course should be used. Students could be exposed to enough of packages, exception handling, generics, and private types so that they can apply these software engineering principles in early simple

176

programming assignments, then apply them in more depth later.

(5). The team projects should be given no later than two thirds of the way through the semester, giving the students adequate time to do them. The topics that are implemented in the projects can be taught in parallel with project development. With the spiral approach, the students have had some program experience with the major software engineering principles by project time.

The author welcomes constructive suggestions from others experienced in teaching similar courses. View graphs of this presentation and sample student program listings are available on request.

References

[1]. ANSI/MIL-STD-1815 A, Reference Manual for the Ada Programming Language , 1983.

[2]. Barnes, John, Programming in Ada , 2nd edition, Addison-Wesley, 1984.

[3]. Booch, Grady, Software Engineering with Ada , 2nd edition, Benjamin-Cummings, 1986.

[4]. Bray, Gary and Pokrass, David, Understanding Ada, A Software Engineering Approach , John Wiley, 1985.

[5]. Mers, Robert C., "Experiences of Pascal Trained Students in an Introductory Ada Course", Proceedings of the 4th Annual National Conference on Ada Technology , 1986.

[6]. Softech Inc: CENTACS Summer Program , U. S. Army (CENTACS), 1983.

[7]. Young, S.J., An Introduction to Ada , 2nd edition, John Wiley, 1984.

This page left blank intentionally

ADA® EDUCATION AND THE NON-COMPUTER SCIENTIST

Dr. Charles C. Kirkpatrick

and

Dr. Paul B. Knese

PARKS COLLEGE

OF

SAINT LOUIS UNIVERSITY

Cahokia, Illinois 62206

May, 1987

179

ADA® EDUCATION AND THE NON-COMPUTER SCIENTIST

Dr. Charles C. Kirkpatrick and Dr. Paul B. Knese

Parks College of Saint Louis University

ABSTRACT

Numerous lessons have been learned during the startup and offering of an Ada® course at Parks College of Saint Louis University. Major obstacles to be overcome during startup include securing hardware and a compiler which are compatible, selecting a text, and promoting the language as more than just another computer language. The first offerings of the course indicate that students are frustrated with long compilation times and with the use of a compiler which has not been validated. Pascal does not seem to prepare a student for learning Ada any better than does FORTRAN. After overcoming an initial bad "name" the course has become very much in demand. Finally, it is recommended that Ada should not be reserved for computer scientists, but should be considered as a first language for everyone who learns to program.

I. INTRODUCTION

This paper relates the experience of starting an Ada® course at Parks College of Saint Louis University and the lessons learned during the first two offerings. The first section of the paper provides a description of the institution and the experience encountered during course start-up. Readers who simply desire a summary of the lessons learned may begin reading immediately in Section V with only a minor sacrifice of continuity.

II. PARKS COLLEGE AND RELATED CURRICULA

Parks College, located in Cahokia, Illinois, was founded by Oliver L. Parks in 1927. It is the oldest certified institution of aviation in America. In its early years it boasted an aircraft factory in addition to students. During World War II Parks College and its subsidiaries trained one of every ten pilots and thousands of aircraft mechanics. In 1946 Parks College was given to Saint Louis University in order to provide future aviation leaders with a broad and more academic education. Today, Parks is one of eleven colleges and occupies one of the three campuses which comprise Saint Louis University.

Parks offers nine bachelor's degree programs ranging from Aerospace Engineering to Transportation, Travel and Tourism. The curricula all have as their theme aerospace technology and management of aviation facilities. The coursework is designed to educate the students in the rich history of aviation and to provide training in the most current technologies in the aviation industry. Representative of the latter goal is the fact that every student who receives a Bachelor of Science degree from Parks must complete at least one computer science course.

For most major disciplines, several courses in computer hardware and software are required. Computer science courses at Parks include FORTRAN for Aerospace Engineering, BASIC for Aircraft Maintenance Engineering, and Pascal for Avionics. No person taking these courses (or any other computer science course) at Parks is a degree candidate in Computer Science. However, this does not detract from the course content, as all course syllabi are designed according to the ACM curriculum recommendations for Computer Science. All of the courses include numerous programming assignments to acquaint the student with the computer and its use as a problem solving tool.

III. INTEREST IN ADA

From the time Ada first became a real possibility for the aerospace industry several of the professors at Parks had expressed sincere interest in offering that capability. With the introduction of the Avionics curriculum in 1983 the need became even more pronounced. In January, 1985, a proposal for development of an Ada course at Parks was prepared. The course was to become

part of the Bachelor of Science in Avionics curriculum, and was given formal approval in June, 1985. The language appeared to be appropriate for students in avionics, who work with real-time embedded computer systems in aviation electronics. The need for such a course was well known and appreciated since most of our avionics graduates seek employment with aerospace companies who do business with the Department of Defense.

A course syllabus was developed and the certified Telesoft Ada compiler was purchased to run on a Codata 3033 (Unix, 68000). For reasons which are still unknown, the Codata-Telesoft Ada combination would not function properly, and the initial course offering had to be cancelled. In addition to usurping most of the time of several faculty members, this experience gave Ada a bad "name" on the campus which proved to to be very difficult to remove.

IV. COURSE START-UP

In June, 1986 the Janus Ada compiler, version 1.5.2, was purchased. It was understood that this compiler did not implement the complete ANSI/MIL-STD-1815 Ada, but after the previous frustration, it was our opinion that any Ada was better than no Ada. The compiler was run on an IBM PC with two floppy disk drives.

The first offering of the course began the following September. Four senior-level avionics students and one faculty member were active participants in the course. Additionally, one student and one faculty member audited the course. The text chosen was Programming in Ada, J. G. P. Barnes, Second Edition, Addison-Wesley Publishing Company, 1984. A copy of the course syllabus is found in the Appendix.

The computer knowledge and experience of the class covered a broad spectrum. Several had only limited BASIC and FORTRAN experience, while others had some Pascal programming in their background. Regardless of previous experience, all were familiar with simple programming topics such as looping and conditional statements. For some it had been years since their last computer science class. The class grades were determined by programming exercises only. No tests or quizzes were given. A term project served as a replacement for a final exam.

The philosophy of the instructor was to teach effective programming techniques within the context of how the students could use the computer, and Ada in particular, to solve problems which were related to their interests. Principles of software engineering such as structured programming were discussed as the means to develop correct and efficient problem solutions.

The first course was well received and the students' enthusiasm generated interest in a repeat performance. The course was offered again in January, 1987. This time several students from disciplines which included Aerospace Engineering and Meteorology took the course. The course syllabus and textbook remained the same as in the first course offering. Grading was based on program assignments, a final project, and weekly quizzes. The quizzes were beneficial to the students and the instructor, as the students' interest in all topics was higher than in the first offering. We propose that using quizzes as a means to encourage students to study and participate in class is a general phenomenon and not unique to Ada education. The final projects in

the second course offering were generally of higher quality than those in the first course, which probably reflect a better overall knowledge of Ada and software engineering as a result of better study habits by the students.

Due to faculty scheduling problems the third offering of the Ada course was cancelled during pre-registration for the Spring, 1987 trimester at Parks College. However, students petitioned the Chairman of the Department of Science and Mathematics and the course is being offered again. The conclusion to be drawn is that the bad "name" of Ada has been forgotten and it is now a class which students demand.

V. LESSONS LEARNED

This section of the paper is divided into two parts. The first part deals with the experience of starting an Ada course, and the second deals with course maintenance and related issues.

Course Start-Up Lessons

Lesson 1. Ada needs a strong advocate. Administrators and faculty who are not familiar with current developments in computer science see Ada as just another language, and an expensive one as well. If hardware purchases are required in addition to the compiler, the investment per student is often considered too large from an administrative view.

Lesson 2. Selection of a compiler is difficult. We have one machine (Hewlett Packard 3000) which serves our other academic needs, but no Ada compiler that will operate on the Hewlett-Packard is available now or in the near future. This means that a compiler purchase must be accompanied by a hardware purchase.

Lesson 3. Text selection is difficult. Most Ada textbooks are very readable if a computer science background is assumed. However, for the computer science novice these texts are confusing. Specifically, most texts do not give complete program examples. To fill this void, programs written by the instructor were distributed at each class meeting.

Lesson 4. Some deals are too good to be true. It is highly recommended that before a compiler purchase, the quality of the product be verified by another user. Parks was courted as a possible test sight for a compiler by a major vendor, but we quickly backed out when we heard from other users of the compiler that the product was not reliable.

Course Maintenance Lessons

Lesson 5. Non-validated compilers give students too many excuses. Rather than take an in-depth look at their code, students will too often blame the compiler for errors they do not understand. This is especially true if it is not always clear what subset of Ada is employed and what changes from the standard have been made. When

the next release became available, we immediately purchased Janus/Ada version 1.6.1. While the upgraded version was more complete, compilations now took even longer than before.

Lesson 6. Long compile times discourage student experimentation. When learning a language, students need to have the opportunity to experiment and "see what happens" when a section of code is changed and executed. This activity does not and will not occur when the compilation time is too long.

Lesson 7. Make no compiler changes during a course. During the first offering we upgraded to the next version as soon as it became available. This happened early in the course and the students adapted readily. This version also removed many of the annoying bugs of the earlier version. Expecting a similar improvement during the second course offering, we immediately upgraded to version 1.6.2. This upgrade required several other subtle changes to the techniques that the students had adopted, and, in general, caused more troubles than it alleviated. Treat any upgrade as you would any other major change. Check it out thoroughly and implement between course offerings.

Lesson 8. Inexperienced students catch on quickly. Based on our experience, students with little background in computer science have no more trouble learning Ada than those with more extensive experience.

Lesson 9. Pascal is not a prerequisite language. FORTRAN programmers struggled only slightly more than Pascal programmers to learn syntax. Knowledge of enumerated types, records, and other topics found in Pascal did not appear to be an advantage. As a matter of fact, it does not appear now that there is any "best" prerequisite language; Ada is complete in itself.

Lesson 10. BASIC and FORTRAN programmers are sometimes more responsive than Pascal programmers. Having been frustrated with a limited set of data types and programming with GOTOs, these programmers were excited about the Ada strong typing and control structures.

Lesson 11. Learning to program in Ada is more than learning a new syntax. Ada forces the programmer to organize ideas before the program is compiled. As a result, Ada becomes part of the solution to the problem rather than becoming another problem in itself.

Lesson 12. Students need tests to learn. Without the testing process, students tend to learn only what they need to know in order to complete their programming assignments.

Lesson 13. Ada education is not only for computer scientists. Students in this class use the computer as a tool to solve problems. Their knowledge of Ada allows them to be experts in using the tool.

184
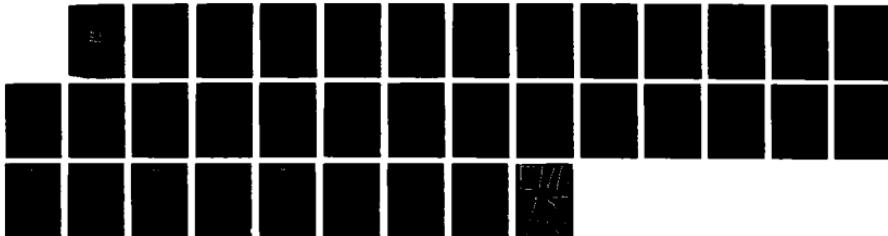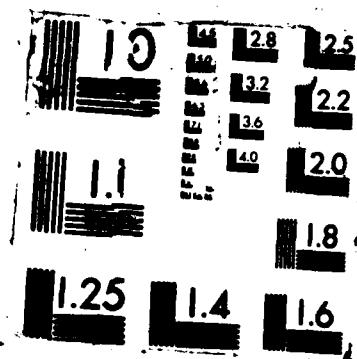
VI. CONCLUSION

Although there were many initial obstacles to be overcome in the development phase of the Ada course at Parks College, the course has been well accepted by the students. Interest is high enough among other students to offer the course nearly every trimester. In the immediate future, there are two major priorities: 1. Decrease the compilation times of Ada programs, and 2. Work towards the goal of obtaining a validated compiler. To achieve both of these goals, funding for hardware is required. The means to reach these goals will most likely be part of a larger solution to the academic computing needs at Parks College.

At the First Annual ASEET Symposium in June, 1986 attention was focused on Ada and what it could do for computer scientists. We have demonstrated that Ada is for anyone who wants to program. There is nothing in the language that makes it any more difficult to learn as a first language than Pascal. Often, BASIC and FORTRAN are taught as first languages. It is generally accepted that Ada makes program development easier and less error-prone than these languages, so it makes no sense to teach the "hard" language or teach the "wrong" language, and then teach Ada later. We postulate that Ada is suitable for instruction as a first language, not merely a language reserved for the privileged use of computer scientists.

If Ada is to be accepted as a major language on the order of FORTRAN or COBOL, we must educate as many people as possible. Ada should not be reserved for the computer scientist simply because it can be complex. All languages deal with complex topics which are best addressed in upper-level computer science courses, but this does not prevent us from using them on the elementary level. We propose that Ada be used as a language to show students that they can master the problem of software development, and use this result to promote the use of computing as an effective and efficient problem solving tool in all disciplines of study.

APPENDIX

952-200 SOFTWARE ENGINEERING FOR EMBEDDED COMPUTERS

I.   Introduction

    A.   Software Engineering

    B.   Software Lifecycle

    C.   Development of ADA®

    D.   Principles of Software Engineering

    E.   Design Methodologies

        1.   Top-down

        2.   Data Structure

        3.   Object Oriented

    F.   Object Oriented Design

II. ADA Concepts

III. Lexical Style

IV. Scalar Types

    A.   Declarations and Assignments

    B.   Scope of Objects

    C.   Types, Subtypes, and Expressions

        1.   Integer

        2.   Natural

        3.   Positive

        4.   Enumerated

        5.   Boolean

        6.   Character

This page left blank intentionally

ADA[1] IN UNDERGRADUATE CURRICULUM AT SAINT MARY COLLEGE

Victor A. Meyer
Department of Mathematics and Computer Science
Saint Mary College
4100 South Fourth Street
Leavenworth, Kansas 66048-5082

Saint Mary College is a small liberal arts college situated on one of the many gently rolling hilltops that surround the community of Leavenworth, Kansas. Founded by the Sisters of Charity of Leavenworth, the college has served the educational needs of northeast Kansas continually for over sixty years. Since its founding, the college has tried to maintain a curriculum that met the needs of the students in an ever changing environment.

The computer science program was added to the college curriculum in 1977 with just a single course being offered. In 1981, the computer science major was offered for the first time. Today the Department of Mathematics and Computer Science offers eighteen computer-related courses. Principle to these courses is the Ada programming language.

Goals

The primary goal of the computer science program at SMC is to provide quality programmer/analysts to the community through its graduates. While junior colleges and technical schools taught the rudiments of programming, it was felt that these programs did not teach the skills necessary to design and implement well structured, documented, and maintainable computer programs. To this end, SMC emphasizes software engineering principles from the first day of the introductory programming courses. In the upper-level courses, critical thinking is emphasized to bring out the student's own creative abilities in designing and implementing computer software.

A second goal of the computer science program is to use state-of-the-art hardware and software whenever possible so that the computer knowledge of our graduates is not obsolete the day they graduate from SMC. Attempting to accomplish this goal on the limited budget available to a small liberal arts college has been an interesting challenge, but not without successes. The college administration, realizing that state-of-the-art equipment and software is necessary to maintain a competitive edge with other schools teaching computer science, has released sufficient funds to outfit a computer science laboratory. With these funds, the department was able to purchase IBM PC/XT[2]

---

[1] Ada is a registered trademark of the United States Government (Ada Joint Program Office)

[2] IBM PC and IBM XT are registered trademarks of International Business Machines Corp.

microcomputers, WordStar[1] professional wordprocessing software, R-Base 4000[2] relational database software, and a site license for the JANUS Ada "D"-Pak[3].

A final goal of the computer science program is to provide for the needs of the local area in terms of computer science professionals. With the presence of Ft.Leavenworth, there are many government contractors searching for good Ada programmers. Many businesses in the area, including SMC, have a need for good BASIC programmers. Area high schools look to SMC to provide Pascal during the summer for their teachers. To accommodate these varying needs, the department allows students to concentrate in any one of these languages at the lower-level. Then, in the upper-level courses, algorithms are taught in pseudocode and the students are expected to apply this pseudocode to their own specific language.

Why Ada?

A fundamental question that needs to be answered is why Ada was chosen as an introductory language in the undergraduate computer science curriculum at SMC. Two of the reasons have already been stated – the desire for state-of-the-art software and the Ada programming needs of nearby Ft.Leavenworth. But are these sufficient reasons to run counter to the current thinking in the academic community which states that Pascal should be the language taught in colleges and universities?

In the 1970's, FORTRAN was the language taught in most colleges and universities. As a result, the work force was flooded with FORTRAN programmers. These programmers recommended to management that FORTRAN be used to develop new software systems. It is only natural for people to recommend a language which has already proved itself capable of being used in large systems and which the programmers feel comfortable using. In fact, it takes a special breed of person to recommend a language which theoretically should be right for the task, but which the person has little knowledge or practical experience.

The result of teaching FORTRAN in the colleges and universities, therefore, was a great proliferation of large software systems written in FORTRAN. The lesson to be learned is that if Ada really is the language for developing large systems, then it is up to the colleges and universities to teach Ada so that programmers have experience in the language and feel comfortable using the language.

Another reason for teaching Ada in colleges and universities stems from the evolutionary development of programming languages. In the early days of computer science, FORTRAN and COBOL were taught as introductory languages because they were the only standardized high-level languages available. They were the "abacuses" of programming. But just as abacuses have evolved into

---

[1] WordStar is a registered trademark of MicroPro International Corp.
[2] R-Base 4000 is a registered trademark of Microrim, Inc.
[3] JANUS Ada "D"-Pak is a registered trademark of R.R. Software, Inc.

mechanical calculators and then into electronic calculators, so have programming languages evolved into bigger and better languages. As the saying goes, we learn from our mistakes. After each new product is built, we begin to see problems with it which ultimately leads us to build an even better product.

Our experience from writing FORTRAN programs showed us a need for structuring programs so that the algorithms could be understood by other people. The principles of "structured programming" were developed to counter bad programming style in languages such as FORTRAN. Attempts were made to change FORTRAN to meet the principles of structured programming (e.g. FORTRAN77), but to maintain compatibility with the earlier standard (i.e. FORTRAN66), it was necessary to leave in undesirable features of the language. The best the educators could do was teach the principle of structured programming and force the student to use these principles when writing class projects. However, once programmers graduated from school, there was no longer a force which kept programmers using structured programming and many programmers reverted back to poor style because they were only interested in getting the program to work.

Nobody doubts that Pascal was a vast improvement over the earlier FORTRAN language. The structure of Pascal itself forced programmers to write good structured programs. This pleased educators because they now felt comfortable students would continue to write structured programs well after their college years.

Years of writing Pascal programs, however, showed the need for more improvements. "Data abstraction", "modules", "information hiding" became the new buzzwords of computer science. Pascal did not support these features. Hence there was a need to develop another new language - Modula-2. Still more programming concepts were discovered (or re-discovered) such as "exception handling", "generics", and "concurrent programming". It became evident that Modula-2 was becoming out-of-date and that another new language needed to be developed. The Department of Defense, in response to years of haphazard programming, filled this need by developing the Ada language.

But what does this evolutionary process have to do with teaching Ada at the undergraduate level? Many colleges and universities teach Pascal in the lower-level and then go on to teach concepts such as data abstraction, exception handling, and concurrent programming in the upper-level courses. The student, however, cannot properly implement these concepts in class projects because Pascal does not support them. At best, the student can only simulate concepts such as information hiding. Some schools respond to this problem by making the student learn a second language, such as Modula-2. But the remainder of the Modula-2 language is so similar to Pascal that one has to ask why Modula-2 wasn't taught from the beginning at the introductory level.

At SMC, the decision to teach Ada was based on this very problem. We wanted the student to learn state-of-the-art programming, including all of the modern programming concepts. We were not satisfied with just teaching the principles on a theoretical level, but we wanted the student to get actual hands-on experience employing these concepts in their class projects.

This is the only way we could feel comfortable that students would continue to employ these concepts after they left SMC. The only standardized language available which supported these modern concepts was Ada.

Does this mean that Ada is the perfect language to be used on the under-graduate level? Definitely not. It only means that SMC has found no better language which incorporates all the current programming concepts. The evolutionary development of programming languages will continue. Ada is only the next stepping stone. Already in the classroom, we are seeing ways that the language can be improved (see "Lessons Learned"). When the next new programming language is developed, SMC will again have to examine the merits of the new language and decide whether a change is in order.

Some of our critics argue that we should not be teaching Ada because very few "help wanted" ads are for Ada programmers - most are for BASIC, CO-BOL, RPG, and FORTRAN. Our response to this criticism is that a graduate who knows Ada is going to have a much easier time learning to program in FORTRAN (or even Pascal) then vice verse. Most of the programming concepts employed in FORTRAN and Pascal are employed in Ada, however there are many new concepts implemented in Ada that are not present in either FORTRAN or Pascal. Hence a graduate who knows Ada does not have to learn any new programming concepts - he only has to learn the syntax and features of the older language.

Another criticism for teaching Ada on the undergraduate level is that Ada is such a complex language that students cannot comprehend all the features of the language. We agree that Ada is a complicated language, but this is only one more reason for starting the student early in learning Ada. In the introductory courses, we make no attempt to teach the entire language. Rather, we teach a subset of the language that is roughly analogous to Pascal. Now, as the student progresses to the upper-level courses, his understanding of Ada can continue to grow as the higher level concepts are presented and the student writes software that uses these concepts. By the time the student graduates, he has had many opportunities to practice the various features available in Ada.

For the critics who say that Ada should be taught as a second language, our response is that students should only learn more than one language when the languages are significantly different from each other. It is important for students to learn the differences between procedural languages (e.g. Ada), functional languages (e.g. LISP), low-level languages (e.g. "C"), fourth generation languages (e.g. Natural), and assembly language (e.g. IBM 360). But to make a student learn Pascal, Modula-2, and Ada in different courses is putting an unnecessary burden on the student. We would prefer to see the student learn one language well, then know bits and pieces of many similar languages.

One criticism of Ada on the undergraduate level that does make sense is that hundreds of students trying to compile Ada programs can bring a large mainframe computer to a standstill. For this reason, large universities may need to limit the use of Ada to their graduate students where class sizes are

smaller. For SMC, as well as other small colleges, this situation does not occur. We have, at most, twenty students in any one computer science class. We have one IBM XT microcomputer (with 640K of memory and hard disk drive) dedicated to Ada compilations. Six other IBM microcomputers are available for editing and testing Ada programs. So far, we have had no problems with overtaxing the systems.

In conclusion, the reasons why SMC decided to teach Ada on the under-graduate level are straight forward. There is a need for Ada programmers in our community because of Ft. Leavenworth. Teaching Ada on the undergraduate level allows us to teach a state-of-the-art language which employs all the concepts of modern programming. As the student moves from lower-level to upper-level courses, the student can continue to learn features of the Ada language as the higher level concepts are taught. Our Ada graduates can eas-ily adapt to other languages because most other languages in use are not as sophisticated as Ada. And finally, the small size of SMC makes it feasible to use Ada in the undergraduate curriculum.

The Ada Curriculum

As mentioned earlier, not all computer science majors are required to study the Ada language. A student can get a computer science degree from SMC knowing either Ada, Pascal, or BASIC. Obviously, we would not recommend to anyone planning to go on to graduate school that they study only BASIC, but there are many area businesses and school systems that are looking for good BASIC programmers. As it turns out, most computer science majors recognize the power of the Ada language and study it at some point during their years at SMC.

There are five primary courses in the curriculum which allow the student to gain experience in writing Ada software. General Programming I & II are lower-level courses which serve as our introductory Ada programming courses. On the upper-level, Data Structures and File Constructs are programming-intensive courses required of all majors which move the student to an intermediate level of programming. These courses are taught using pseudocode and the students are allowed to implement their programming projects in either Ada, Pascal, or BASIC. Software Engineering is also an upper-level course where groups of students must work as a team to develop a software system. The teams are free to implement their projects using either Ada, Pascal, BASIC, COBOL, or the R-Base 4000 relational database system.

As mentioned, General Programming I & II serve as our introductory Ada programming courses. However, we deliberately named the course "General Pro-gramming" because we did not want to permanently attach any specific program-ming language to the course. The purpose of the course is to teach program-ming concepts independent of any specific language. However, since this is often the student's first exposure to computer programming, we do apply the concepts to a specific programming language and all examples are written in that one specific language. When the courses were originally developed, Pascal was the specific language used to demonstrate programming concepts.

In the last two years, Ada has been used. In the future, it is very possible that some other language may be used.

General Programming I is designed to take a student who has no programming experience and teach that student the control structures of modern programming (i.e. sequence, decision branching, looping, procedures, functions, and exception handling). Only the elementary data types (i.e. integer, floating point, boolean, character, and enumeration) are taught, but proper declaration of types and subtypes are fully covered. Flowcharts are used to design the simple algorithms that are taught at this level and the students are introduced to the use and purpose of the various libraries that are present in JANUS Ada.

Programming projects in General Programming I are designed to introduce the students to the principles of software engineering. The project is assigned in the form of a small requirements document and users manual. Normally the requirements document is organized as a set of functional requirements followed by a set of non-functional requirements. The simple users manual is normally in the form of a sample run. Great emphasis is placed on accurately following the stated requirements and sample run.

Normally, students have one week to complete each project. A flowchart showing the design of the project is due by the next class meeting. During the remainder of the week, the students must implement the design. If the design is correct, but the student's program does not accurately follow the design, then points are deducted. Experience has shown us that it is not a good pedagogic practice to allow the student to turn in the design and the project together. Inevitably, the student will first write the program and then draw the flowchart based on the completed program.

The number one factor for grading a student project in General Programming I is how well the finished program can be understood by someone else. Internal program comments, conscientious naming of identifiers, and simple algorithms through modularization are deemed very important to accomplishing this goal. Efficiency is considered important, but not as important as writing a program that is "readable". Once exception handling is introduced, then proper error checking is also emphasized on student projects.

In General Programming II, the student is introduced to the data structuring mechanisms of modern programming (i.e. records, arrays, pointers, and files). Modules are introduced as a means for accomplishing data abstraction through the principle of "information hiding". Pseudocode becomes the principle vehicle for writing algorithms. Classroom examples show how a properly written program can be quickly modified to meet the needs of a changing environment.

The nature of student projects also change. Projects are significantly larger with fewer projects assigned and more time allotted to each product. Students are required to design the projects using pseudocode. Emphasis is now placed on writing programs that are "universal" in nature and maintainable.

General Programming II is given a 300-level number in the college catalog. This means that the course is taught as a lower-level course, but that a student can get upper-level credit by completing some extra work that is worthy of upper-level credit. For General Programming II, a student is given upper-level credit if he successfully designs and implements a practical addition to the Ada library. Normally, this involves writing an Ada package which allows other students to interface with a new terminal or printer. However, if the student wishes to try for "honors-in-course", the project can easily be expanded to involve implementing an exotic abstract data type.

As the student moves into the File Constructs course, he now comes into contact with students from varying programming language backgrounds. No longer are lessons geared to only one language. The principles of sequential and direct file accessing are presented in pseudocode and students are expected to apply the pseudocode to their own specific language. Projects are much larger in nature. For many students, the course can be a rude awakening because, for the first time, the instructor is not telling them how to implement the projects directly in their own language.

The next course encountered by the student is Data Structures. In the department, we have decided that this course will be the one where critical analysis skills are emphasized. Again, lessons are taught using pseudocode. But now, the students are not given cookbook style algorithms for solving problems. Instead, students are given examples of some of the algorithms necessary for implementing abstract data structures such as stacks, queues, and trees. Then, the students are expected to develop the remaining algorithms on their own. What we hope to accomplish is the awakening of the students own creative ability for developing algorithms in new and challenging situations.

The student projects for Data Structures are similar in length and complexity to those of File Constructs. Students writing their projects in Ada are expected to write their abstract data types using generic packages with appropriate information hiding. A major emphasis of these projects is to write modules that can be replaced easily by improved versions of an abstract data type without having to make changes in the rest of the system.

Software Engineering is a course where students can gain additional experience writing Ada programs. In this course, the emphasis is placed on group interaction. The entire class is divided into groups of three to five students each. The groups are required to interact with a user (usually the instructor) and write a Software Requirements Document, a User's Manual, a Design Document (which includes data flow diagrams, structure charts, IPO charts, and pseudocode), and then implement the design in the language of their choice. Even though the class is open to students with any programming language background, much of the textbook [Sommerville, 1985] and course content is geared to the Ada language.

One last course rounds out the Ada part of the curriculum. Special Top-

ics is a senior level course that can be used to teach other aspects of computer science not covered by any other course. There are plans to occasionally use this course to teach concurrent programming concepts to the students using Ada's tasking mechanism. Unfortunately, our JANUS Ada compiler does not currently support tasking and there is no indication when tasking will be implemented in JANUS Ada.

Lessons Learned

Since the teaching of Ada in the undergraduate curriculum is only in its second year, it is not yet possible to determine if our long range goals are going to be successful. Students are just beginning to enter the upper level courses after having taken Ada in General Programming I & II. However, since the use of Ada in the curriculum is new, there are many opportunities to learn what works and what does not work. The following is a compilation of some of the lessons that we have learned.

So far, we have had no major problems with having adequate equipment available for our Ada students. With one IBM XT computer dedicated to Ada compiles and six IBM PC computers available for editing and testing Ada programs, there have been few people waiting to get on a computer. Again, this is the advantage of a small college. Ada compiles take about three minutes on the IBM XT and students have accepted this amount of time as a fact of life.

Since Pascal has not become widely accepted outside of educational institutions, very few graduates with baccalaureate degrees in computer science are actually finding Pascal jobs. Only one SMC graduate is known to have landed a Pascal programming job. Many of our better students are actually employed writing Modula-2 software for Department of Defense contractors. A couple of our graduates have returned to SMC after graduation for the purpose of upgrading themselves to Ada to improve their job prospects (one of them was advised by an employment agency to learn Ada). The lesson to be learned is that graduates who know Ada are in a better position to find jobs than graduates who know Pascal.

To date, we have taught the General Programming I course twice with Ada. We have found that students comprehend Ada the same as they comprehended Pascal. Even though Ada is a more complex language, not one of the students felt that Pascal should still be taught in the course. In many cases, class examples were actually simpler and easier to comprehend by the students because of the features available in Ada. Obviously, a major part of this success is due to the fact that we do not try to teach every detail of the Ada language. The lesson to be learned is that Ada is eminently suitable as an introductory language.

Since the concepts presented in General Programming I are implemented similarly in both Pascal and Ada (exception handling being the major exception), we allow students who have had a semester of Pascal programming to go directly into General Programming II. The first two weeks of General Pro-

gramming II are spent reviewing the concepts learned in General Programming I. The students who took General Programming I find the review useful and the Pascal students get a good orientation into the Ada language.

While this option has only been tested once, we have learned that the option works. Pascal students had no problems adapting to Ada at this level. In fact, these former Pascal students became our strongest supporters of Ada and encouraged other students to learn Ada instead of Pascal. Those students who went on to take other programming-intensive courses chose to do their programming projects in Ada.

The only problem that we encountered with these former Pascal students was that they tended to write Ada programs with a strong Pascal dialect (e.g. always exiting out of a loop at the beginning or end rather than exiting the loop where it was most appropriate). The lesson to be learned is that students continue to use the techniques that they learned in their first language and have to be retrained to think in the new language.

This situation is not unlike a person who learns a foreign language. It is not difficult to identify a person whose native language is not English - their accent gives them away. In fact, their accent often identifies that person's native tongue (i.e. we can tell that a person is French, German, Italian, etc.). Any language teacher will tell you that in order for an American to speak French properly, he must think in French and not just translate between English and French.

The same principle is true in programming languages. A person whose first programming language is Pascal learns to think in Pascal. If the person then tries to write Ada programs, the syntax will be Ada, but the thought process will still be Pascal. This situation will continue to exist until the person divorces himself of Pascal and learns to think in Ada. The bottom line is that Ada should be taught as the student's first programming language rather than as a second language. We do not see where a student gains anything by first learning Pascal and then learning Ada.

The problem of people learning Ada after they have learned Pascal manifests itself in Ada textbooks. Most Ada textbooks are prefaced by the statement that they are intended for students who have already learned a high-level programming language such as Pascal. Also, the authors bias towards Pascal often comes through in the examples used in the textbook. Another problem is that Ada concepts that are significantly different from Pascal are often relegated to the end of the book. Our current textbook [Saib, 1985], for example, teaches exception handling in chapter nineteen of a twenty-one chapter book. Exception handling is not a complicated process and students should learn the proper way to do error checking much earlier in the course.

Another problem resulting from people with Pascal backgrounds is that they take the attitude that Pascal contains everything that is really important in modern programming and there is no need to teach any other language at the introductory level. This problem surfaced when SMC applied for a Na-

tional Science Foundation grant to upgrade their equipment and software to include Ada as the introductory language. Five of the seven reviewers of the proposal stated outright that they did not feel Ada should be taught at the introductory level – most cited Pascal as the better choice. The remaining two reviewers questioned the feasibility of teaching Ada from the standpoint of hardware availability. Needless to say, SMC ignored the recommendations of the reviewers and the college administration came through with the necessary funds to put Ada into the curriculum.

No language is perfect and Ada definitely has its problems. BASIC and FORTRAN programmers must be taught to discipline themselves to write good structured programs. The beauty of the Pascal language is that it's features force the programmer to use good structure in the program – but this was only learned by years of bad programming in BASIC and FORTRAN. Now Ada is trying to implement new programming concepts. At SMC, we are finding that we must again teach the students to discipline themselves to use these new features properly as Ada does not force proper usage of them.

A primary example of this problem is Ada's feature for declaring "new" types (e.g. new integer) for establishing incompatibilities between variables of similar types. We believe the concept to be a good concept and we strongly emphasize proper declaration of types and ranges. However, students quickly learn that programming in Ada is much easier if they ignore this feature and declare all their whole number variables to be of type "integer" irregardless of whether the real world entities represented by these variables are compatible with each other or not. As a result, we find ourselves trying to make the student discipline himself to use the feature properly.

Someday, hopefully, someone will discover a better way to set up incompatibilities between similar data types which will force the programmer to use the feature properly. At least Ada pioneered the concept and we can teach the concept in the classroom and have the student practice the concept in their programming projects. If Pascal was used in the classroom, we would only be able to teach the concept. But this would be like teaching a mathematics course and not requiring the student to do any homework. Experience has shown us that a student really doesn't learn a concept until he has to sit down and apply the concept himself in a homework or test situation. With Ada as our introductory language, we can make the student apply the concept in a homework project.

One final lesson learned from adding Ada to the curriculum is that college instructors often have to teach themselves how to program in Ada. Courses in Ada are not readily available in universities and corporate workshops often put a significant financial burden on instructors from small liberal arts colleges. At SMC, one of the two principle computer science instructors is self-taught in Ada and is teaching Ada to the other instructor. Being that JANUS Ada is not a validated compiler, there are some important features not yet implemented which make it difficult for the instructors to practice writing true Ada programs. Even though we have been continually expanding SMC's library collection of Ada literature, there is no substitute for actual experience in writing Ada software.

Conclusion

At SMC, we are convinced that Ada can and should be taught on the introductory level. Despite warnings to the contrary, SMC forged ahead and added Ada to the curriculum. So far, the warnings have proven to be inaccurate and experience is showing us that we made the right decision.

In our opinion, the only good reason for not teaching Ada on the undergraduate level is the problems that a mainframe computer will have when suddenly besieged by large numbers of students simultaneously compiling Ada programs. Hopefully, the future holds the promise for smaller and more efficient Ada compilers which will alleviate this problem. Until then, the future of Ada in the undergraduate curriculum may very well be limited to small colleges, such as Saint Mary College, where student numbers are low and the faculty realizes Ada's potential for state-of-the-art computer science education.

Bibliography

[Saib, 1985]   Sabina Saib, Ada: An Introduction, CBS College Publishing, New York, NY (1985).

[Sommerville, 1985] I. Sommerville, Software Engineering, Second Edition, Addison-Wesley Publishing Co., Reading, MA (1985).

This page left blank intentionally

THE PROGRAMMING TEAM AND THE ACCELERATED COURSE AS METHODS FOR TEACHING ADA

David L. Barrett
Department of Computer Science
East Texas State University
Commerce, Texas 75428

Abstract

Results obtained by a one semester Ada programming project demonstrated that an accelerated short course in the elementary *features of the Ada* programming language could be incorporated into an existing survey of programming languages course, and produced a number of other observations and conclusions applicable to the development of more extensive instruction in Ada and the principles of software development.

I. Introduction

The definition of limited and, therefore, widely attainable objectives for education in Ada may present challenges equal to those faced in implementing more extensive programs. Universities, which include schools of engineering and which conduct large programs of research in defense and aerospace fields, have introduced extensive curricula in Ada and software engineering. The strategies that those universities have adopted for the introduction of Ada and related areas of study may not always be appropriate for other institutions, which may be smaller, have commitments which must take precedence over Ada instruction and research, or have facilities and budgetary limitations which preclude the rapid, large-scale introduction of instruction in Ada and software engineering into their computer science curriculum.

In 1986 the Department of Computer Science at East Texas State University initiated a program to expand and enhance its curriculum and computing facilities. The primary objective of the Computer Science Department was to attain full compliance with the CSAB criteria for accrediting programs in computer science at the earliest feasible date. Another objective of the department was to develop additional courses in specific topics relevant to the professional interests of its students.

In recognition of the growing importance of the Ada programming language, a decision was made to incorporate instruction in Ada into the computer science curriculum at the earliest feasible date. As a provisional measure, a compiler which supported a subset of Ada was purchased to permit students and faculty to begin to familiarize themselves with Ada pending the acquisition of a validated compiler. During the fall semester of 1986 a special topics course on Ada was organized for a limited number of graduate students by Visiting Professor L.C. Harrison, who was then the faculty member provided to East Texas State University by E-Systems. Toward the end of the semester, the department initiated action to obtain a validated compiler,

which had been identified by a study undertaken as part of the special topics course to be suitable for use in instruction and in program development.

Although steps had been taken to obtain a satisfactory Ada programming environment, concern about the eventual date of acquisition of the Ada compiler and short-term limitations on the number of possible users of the computer on which the compiler would be run led to the realization that conducting a lecture course in Ada would not be possible during the spring semester of 1987.

The department concluded that it would be expedient to continue the study of Ada on the basis of work in special topics courses, which would be made available to undergraduate as well as graduate students. Suggestions for the conduct of study in Ada for the spring were requested by Professor Harrison from his graduate students, and the concept of organizing the students enrolled in the classes into a programming project was adopted. The project was to be primarily a student undertaking. Management of the project and the preparation of project documents, including the final report on project activities were assigned to the author of this paper, a graduate student. Overall evaluation and supervision of project activities would be undertaken by Professor Harrison as instructor of the two courses.

II. Project Goals

The organization of the special topics classes into a programming project created the opportunity to investigate a number of topics of interest to the department.

First, the project would provide data which might be applicable to the design of a course or a sequence of courses of instruction in Ada. A software design and development course, based on ACM recommended course CS 14, is planned for introduction in the spring of 1988. The course outline specified for CS 14 provides for a team project involving the organization, management, and development of a large scale software project by students working in teams. Experience gained through the activities of the programming teams during spring, 1987, would contribute to the decision as to whether Ada, or another language, should be employed in the projects course, and it would provide a test of the validity of the programming team as a vehicle for instruction in the effective use of the Ada programming language.

Second, as none of the undergraduate participants in the software project had previous experience in Ada programming, it would be necessary to give them instruction in the fundamentals of Ada syntax before the commencement of the software development phase of the project. Experience gained by the graduate students, who participated in the special topics course during fall, 1986, indicated that a short course, which covered a Pascal-like subset of Ada could be completed successfully in a period of from three to four weeks. Although such an accelerated course could not provide a complete knowledge of Ada, it would provide the students with a foundation sufficient to participate effectively in software development as members of programming teams. The development of an Ada short course for the project also would

204

facilitate the introduction of Ada as one of the languages covered by an existing undergraduate level survey of languages course, which compares the features of several programming languages to illustrate the significant features and underlying concepts of algorithmic languages.

Third, although no immediate plan for the replacement of Pascal as the principal language of instruction for introductory courses was contemplated, the development of an Ada short course during the spring semester of 1988 and its implementation as a component of the existing survey of language course would provide the beginnings of a body of experience in instruction in Ada that support such a transition if, subsequently, it were deemed to be desirable.

Fourth, the conduct of the project exercise and the introduction of Ada as a part of the undergraduate survey of languages course would begin the development of a group of students with a knowledge of Ada sufficient to permit them to participate in future projects or more advanced courses involving the Ada programming language and the techniques of software development which Ada supports.

III. Project Organization

The graduate and undergraduate classes followed different tracks. During the first four weeks of the spring semester, undergraduate students attained an elementary knowledge of the Ada programming language through participation in the Ada short course. Graduate students worked to improve their knowledge of Ada and were assigned roles in the project organization. After the completion of the Ada short course by the undergraduate students the graduate and undergraduate groups were merged into the unified project organization.

Initially, a project organization with one graduate and two undergraduate teams was projected; however, it became apparent before the beginning of the spring semester that several of the graduate students would be needed to perform roles other than that of programmer.

The organizational structure of the project finally adopted for the project was hierarchical. Three programming teams consisting of several undergraduate students and one graduate student were organized. In each team the graduate student was assigned the role of lead programmer. One graduate student was assigned the role of configuration manager for the project and another the role of project analyst, both of these positions, particularly that of configuration manager proved to be extremely helpful to the design and development of the software produced by the project. The project also had to provide its own system operator for the MicroVAX on which its Ada compiler would be installed, and a system operator was named from among the graduate students.

As the organizational structure adopted in January, 1987, proved to be adequate, it was retained throughout the semester, although several of the programmers were redistributed among the programming teams in March according to the assignments that were given to each team at that time.

205

IV. Conduct of Project Activities

Concern over the date of availability of the department's validated Ada compiler had been a significant factor in the decision to proceed with the programming project as an alternative to a standard lecture course. That concern proved to be well founded and it was not until March, after much uncertainty, that the compiler was delivered and not until early April that the programmers were able to use the compiler effectively. As a result, most of the programming undertaken by the project was done using the subset compiler. The original software development goal of the project to develop a multi-tasking package for possible subsequent use in implementing concurrent multi-processing of signal data was not pursued during the spring semester, and the final programming activity of the programming teams was largely a minor extension of an earlier exercise which had been utilized to familiarize the programmers with Fourier transforms in anticipation of developing the multi-tasking package. Although the limitation of the scope of project software development was disappointing, the more important goal of the project to explore methods of instruction for Ada was not considered to have been compromised.

The conduct of the short course in Ada was not severely affected by the necessity of utilizing the subset compiler. The progress of the short course was observed by the instructor of the undergraduate survey of languages course and, as the results of the short course were good, she decided to incorporate it as a part of her course and taught it to her students later in the spring semester with favorable results.

In addition to the testing and validation of a short course in Ada, insight was obtained into the usefulness and validity of the programming team as a method of instruction for the Ada programming language, which were submitted to the Department of Computer Science in the project final report.

V. Observations and Conclusions

Several limitations concerning the reliability of data from the observation of team programming activities were accepted in the initial planning of the project. The fact that a scientifically valid experimental or quasi-experimental design for the measurement and evaluation of student performance would not be implemented was accepted as unavoidable. "One-shot" case studies have been stigmatized as being subject to misinterpretation and, therefore, having little scientific value. Reliance on such studies as a minimum reference point may be dictated, however, by external factors which exclude the possibility of utilizing designs which are more valid, but much more complex and difficult to execute. Another factor in evaluating the reliability of the project data is the fact that observation and evaluation of the project's results were conducted by persons participating in the work of the project. Participant observation is not considered to be desirable because of the possibilities of the observer influencing the course of the study or adopting a biased interpretation of its results. As the availability of the validated Ada compiler was delayed, the development of statistical data on the project was not attempted and evaluation of the work of the project was based on subjective criteria.

Because of the recognized limitations of the project study, its conclusions are considered preliminary and subject to subsequent revision. The results of subsequent software development and instruction in Ada will be monitored by the department and correlated with the project results.

The incorporation of limited instruction in Ada into the undergraduate survey of languages course was successful and its success provided the first confirmation of a conclusion derived from the work of the project.

Other observations and conclusions of the project, which have been submitted to the Department of Computer Science but which have not yet been confirmed by additional research are that

1.   Although the suitability of Ada for introductory computer science instruction was not directly investigated, the relative ease with which the Pascal-like subset of Ada was taught as an accelerated short course to computer science students with a previous introduction to Pascal suggests that it could be taught successfully in a semester-long introductory course in computer science, provided that introductory texts based on the Ada programming language were available.

2.   The validity of the programming team as a means of instruction in the advanced features of Ada depends upon the computer science background of the students participating in the programming project. Although the features of Ada which resemble Pascal in their syntax and operation can be taught rather quickly, it appears that advanced features will be learned more slowly. Based on a single trial involving instruction in the rudiments of tasking, it would appear that a course of study lasting at least one semester is required to prepare students for software development that involves the use of the more complex features of the Ada programming language. If a participant in a project oriented course already is familiar with the principles of software development, then his, or her, time may be devoted to studying the features of the Ada programming language, which implement those principles. Without such a background the student's time will be divided between learning Ada and the techniques of software development. Possible solutions might be to provide students with a greater exposure to software development techniques in introductory courses, to limit the project to the use of a subset of Ada and emphasize software development techniques, or conversely to emphasize Ada and so structure the project that the participants would need to have only limited knowledge of software development for the completion of their work. The most desirable solution would be to offer an advanced Ada course of three or four semester hours in addition to the project oriented software development course based on CS 14, instruction for which would be exclusively in the area of software development principles and techniques.

3.   The conduct of project oriented instruction in Ada would be facilitated by the existence of toolsets which would support instructional projects. The development of standardized, reusable, project oriented instructional modules also would contribute significantly to the effectiveness of instruction in software development. Such toolsets and modules could be developed by individual institutions for their own use; however, the most effective use of

such modules would be achieved if they were developed by a national institution such as the Software Engineering Institute and made available to any institution which would benefit from their employment.

4. The limited experience obtained through the activities of the Ada software development project conducted during the spring semester, 1987, indicates that many texts and other publications assume a level of cooperativeness and social consciousness among the members of programming teams, which may not always be present in reality. The personnel and management aspects of software development may present challenges more severe than those of software design and implementation, and they represent an area for research, which is potentially as important as that concerned with technical aspects.

Acknowledgement

INDEX

$\mathcal{A}$da®IC

Ada® INFORMATION CLEARINGHOUSE
Ada Joint Program Office

NET Mail: Ada-Information @ Ada-20

The Ada Information Clearinghouse facilitates the transfer of timely information between the Ada Joint Program Office and the Ada User Community.

The Clearinghouse...

o coordinates the collection, integration and distribution of documentation on all aspects of the Ada language and associated aspects of DoD's Software Initiative

o announces recent activities and general information on Ada via Ada-Information, an on-line file accessible via MILNET or TELENET

o provides recent updates on Ada conferences, seminars, classes and textbooks

CURRENTLY...

The Ada IC is seeking active input from the Ada User Community to expand our base of knowledge on current Ada activities in the private sector.

If your organization would like to announce an activity update on:

o compilers
o courses/in-house seminars
o conferences
o publications

Or would like to obtain information on

o How to obtain MIL-STD-1815A (1983)
o CAIS Status
o Compiler Validation Updates
o Education and Training

Contact the Ada Information Clearinghouse at either address listed above.

Form GO2-0885

This page left blank intentionally

$\mathcal{A}da_{\circledR}\mathbb{I}C$

Ada® INFORMATION CLEARINGHOUSE
Ada Joint Program Office

Ada INFORMATION CLEARINGHOUSE REQUEST FORM

Name: ( ) Mr. ( ) Ms. ( ) Other _____

Company: _____

Address: _____

_____

_____

City: _____ State: _____ Zip: _____ - _____

Country: _____ Telephone: ( ) _____ - _____

Autovon: _____

Type of Information Requested

[ ] General Information Packet

This packet includes general information on Ada compilers, Ada related documents and publications, Ada events, and other services of the AdaIC.

[ ] Education and Training Packet

This packet contains a listing and description of upcoming Ada classes and seminars, conferences and programs, Ada textbooks and other related publications, and reprints of general articles on the Ada language.

[ ] Historical Information

This packet contains old issues of the AdaIC newsletter, information on the history of the Ada programming language, and bulletins archived from the "Ada Today" file of items of interest to the Ada community.

[ ] Newsletter Mailing List

The AdaIC Newsletter is produced quarterly and is sent free of charge.

[ ]   CREASE Survey
Please include me for the next survey for the Catalog of Resources for Education in Ada and Software Engineering (CREASE).

SEND ALL REQUESTS TO:

**Ada Information Clearinghouse ● 3D139 (1211 S. Fern, C-107) ● The Pentagon ●
Washington DC 20301-3081**

The Ada Information Clearinghouse is contractor operated for the AJPO.

● Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

213

This page left blank intentionally

AdaIC
3D139 (1211 S. FERN, C-107)
The Pentagon
Washington, D.C. 20301-3081
(703) 685-1477
(301) 731-8894

Ada®IC

Ada® INFORMATION CLEARINGHOUSE
Ada Joint Program Office

NET Mail: Ada-Information @ Ada-20

Public Access to the Ada Information Bulletin Board

The Ada Information Bulletin Board is a publicly available source of information on the Ada language and Ada activities. Sponsored by the Ada Joint Program Office and maintained by the Ada Information Clearinghouse, this Bulletin Board is used to announce current events, general activities and indicate the status of various Ada compiler efforts. Access to the Bulletin Board requires a computer terminal and modem or a personal computer and modem.

The Ada Information Bulletin Board system can be accessed by dialing (202) 694-0215, using a 300 or 1200 baud modem. Users should set their telecommunications package with the following parameters:

        Baud rate = 300 or 1200
        Parity   = none
        data bits = 8
        stop bits = 1

After you are connected, the bulletin board waits to receive three carriage returns so that it can match your telecommunications parameters (The bulletin board initially answers at 1200 baud, no parity, 8 data bits, and 1 stop bit).

Currently, the only directory available is <Ada-Information>. Within that directory, "FILES.HLP" contains an alphabetical listing of all information files available, descriptions of their contents, and an Ada IC point of contact for further information.

NOTES

NOTES

217

NOTES

END

9-87

DTIC